Spring Integration Reference Guide

Mark Fisher, Marius Bogoevici, Iwein Fuld, Jonas Partner, Oleg Zhurakousky, Gary Russell, Dave Syer, Josh Long, David Turanski, Gunnar Hillert, Artem Bilan, Amol Nayak, Jay Bryant

Version 5.3.0.RC1

Table of Contents

Preface . . . 2
1. Requirements . . . 3
1.1. Compatible Java Versions . . . 3
1.2. Compatible Versions of the Spring Framework . . . 3
2. Code Conventions . . . 4
3. Conventions in This Guide . . . 5
What's New? . . . 6
4. What's New in Spring Integration 5.3? . . . 7
4.1. New Components . . . 7
4.2. General Changes . . . 8
4.3. AMQP Changes . . . 9
4.4. HTTP Changes . . . 9
4.5. Web Services Changes . . . 9
4.6. TCP Changes . . . 9
4.7. RSocket Changes . . . 9
4.8. Zookeeper Changes . . . 9
4.9. MQTT Changes . . . 10
4.10. (S)FTP Changes . . . 10
4.11. File Changes . . . 10
Overview of Spring Integration Framework . . . 11
5. Spring Integration Overview . . . 12
5.1. Background . . . 12
5.2. Goals and Principles . . . 12
5.3. Main Components . . . 13
5.4. Message Endpoints . . . 15
5.5. Configuration and `@EnableIntegration` . . . 21
5.6. Programming Considerations . . . 23
5.7. Programming Tips and Tricks . . . 26
5.8. POJO Method invocation . . . 29
Core Messaging . . . 32
6. Messaging Channels . . . 33
6.1. Message Channels . . . 33
6.2. Poller . . . 57
6.3. Channel Adapter . . . 63
6.4. Messaging Bridge . . . 67
7. Message . . . 71
7.1. The `Message` Interface . . . 71
7.2. Message Headers . . . 71

7.3. Message Implementations . . . 76
7.4. The `MessageBuilder` Helper Class . . . 76
8. Message Routing . . . 79
8.1. Routers . . . 79
8.2. Filter . . . 104
8.3. Splitter . . . 108
8.4. Aggregator . . . 112
8.5. Resequencer . . . 135
8.6. Message Handler Chain . . . 139
8.7. Scatter-Gather . . . 143
8.8. Thread Barrier . . . 147
9. Message Transformation . . . 150
9.1. Transformer . . . 150
9.2. Content Enricher . . . 161
9.3. Claim Check . . . 170
9.4. Codec . . . 174
10. Messaging Endpoints . . . 178
10.1. Message Endpoints . . . 178
10.2. Endpoint Roles . . . 192
10.3. Leadership Event Handling . . . 194
10.4. Messaging Gateways . . . 195
10.5. Service Activator . . . 216
10.6. Delayer . . . 221
10.7. Scripting Support . . . 226
10.8. Groovy support . . . 230
10.9. Adding Behavior to Endpoints . . . 234
10.10. Logging Channel Adapter . . . 258
10.11. `java.util.function` Interfaces Support . . . 260
11. Java DSL . . . 264
11.1. DSL Basics . . . 265
11.2. Message Channels . . . 267
11.3. Pollers . . . 269
11.4. DSL and Endpoint Configuration . . . 270
11.5. Transformers . . . 271
11.6. Inbound Channel Adapters . . . 271
11.7. Message Routers . . . 272
11.8. Splitters . . . 274
11.9. Aggregators and Resequencers . . . 275
11.10. Service Activators and the `.handle()` method . . . 276
11.11. Operator log() . . . 277
11.12. Operator intercept() . . . 277

11.13. MessageChannelSpec.wireTap() . . . 278
11.14. Working With Message Flows . . . 279
11.15. FunctionExpression . . . 280
11.16. Sub-flows support . . . 280
11.17. Using Protocol Adapters . . . 284
11.18. IntegrationFlowAdapter . . . 286
11.19. Dynamic and Runtime Integration Flows . . . 289
11.20. IntegrationFlow as a Gateway . . . 292
11.21. DSL Extensions . . . 294
12. Kotlin DSL . . . 296
13. System Management . . . 298
13.1. Metrics and Management . . . 298
13.2. JMX Support . . . 305
13.3. Message History . . . 315
13.4. Message Store . . . 317
13.5. Metadata Store . . . 321
13.6. Control Bus . . . 323
13.7. Orderly Shutdown . . . 324
13.8. Integration Graph . . . 325
13.9. Integration Graph Controller . . . 332
Integration Endpoints . . . 335
14. Endpoint Quick Reference Table . . . 336
15. AMQP Support . . . 340
15.1. Inbound Channel Adapter . . . 340
15.2. Polled Inbound Channel Adapter . . . 348
15.3. Inbound Gateway . . . 348
15.4. Inbound Endpoint Acknowledge Mode . . . 352
15.5. Outbound Channel Adapter . . . 353
15.6. Outbound Channel Adapter . . . 353
15.7. Outbound Gateway . . . 358
15.8. Asynchronous Outbound Gateway . . . 363
15.9. Inbound Message Conversion . . . 368
15.10. Outbound Message Conversion . . . 369
15.11. Outbound User ID . . . 370
15.12. Delayed Message Exchange . . . 370
15.13. AMQP-backed Message Channels . . . 371
15.14. AMQP Message Headers . . . 374
15.15. Strict Message Ordering . . . 377
15.16. AMQP Samples . . . 378
16. Spring ApplicationEvent Support . . . 380
16.1. Receiving Spring Application Events . . . 380

16.2. Sending Spring Application Events . . . 382
17. Feed Adapter . . . 385
17.1. Feed Inbound Channel Adapter . . . 385
17.2. Duplicate Entries . . . 386
17.3. Other Options . . . 386
17.4. Java DSL Configuration . . . 387
18. File Support . . . 388
18.1. Reading Files . . . 388
18.2. Writing files . . . 401
18.3. File Transformers . . . 409
18.4. File Splitter . . . 410
18.5. Remote Persistent File List Filters . . . 414
19. FTP/FTPS Adapters . . . 416
19.1. FTP Session Factory . . . 416
19.2. Advanced Configuration . . . 418
19.3. Delegating Session Factory . . . 420
19.4. FTP Inbound Channel Adapter . . . 421
19.5. FTP Streaming Inbound Channel Adapter . . . 429
19.6. Inbound Channel Adapters: Polling Multiple Servers and Directories . . . 432
19.7. Inbound Channel Adapters: Controlling Remote File Fetching . . . 433
19.8. FTP Outbound Channel Adapter . . . 434
19.9. FTP Outbound Gateway . . . 440
19.10. FTP Session Caching . . . 448
19.11. Using `RemoteFileTemplate` . . . 449
19.12. Using `MessageSessionCallback` . . . 450
19.13. Apache Mina FTP Server Events . . . 451
19.14. Remote File Information . . . 452
20. Pivotal GemFire and Apache Geode Support . . . 453
20.1. Inbound Channel Adapter . . . 454
20.2. Continuous Query Inbound Channel Adapter . . . 455
20.3. Outbound Channel Adapter . . . 456
20.4. Gemfire Message Store . . . 457
20.5. Gemfire Lock Registry . . . 458
20.6. Gemfire Metadata Store . . . 459
21. HTTP Support . . . 461
21.1. Http Inbound Components . . . 461
21.2. HTTP Outbound Components . . . 464
21.3. HTTP Namespace Support . . . 465
21.4. Configuring HTTP Endpoints with Java . . . 476
21.5. Timeout Handling . . . 478
21.6. HTTP Proxy configuration . . . 481

21.7. HTTP Header Mappings . . . 482
21.8. Integration Graph Controller . . . 483
21.9. HTTP Samples . . . 484
22. JDBC Support . . . 486
22.1. Inbound Channel Adapter . . . 486
22.2. Outbound Channel Adapter . . . 490
22.3. Outbound Gateway . . . 493
22.4. JDBC Message Store . . . 494
22.5. Stored Procedures . . . 499
22.6. JDBC Lock Registry . . . 510
22.7. JDBC Metadata Store . . . 511
23. JPA Support . . . 513
23.1. Functionality . . . 514
23.2. Supported Persistence Providers . . . 514
23.3. Java Implementation . . . 514
23.4. Namespace Support . . . 515
23.5. Inbound Channel Adapter . . . 519
23.6. Outbound Channel Adapter . . . 524
23.7. Outbound Gateways . . . 532
24. JMS Support . . . 543
24.1. Inbound Channel Adapter . . . 544
24.2. Message-driven Channel Adapter . . . 545
24.3. Outbound Channel Adapter . . . 547
24.4. Inbound Gateway . . . 548
24.5. Outbound Gateway . . . 549
24.6. Mapping Message Headers to and from JMS Message . . . 557
24.7. Message Conversion, Marshalling, and Unmarshalling . . . 559
24.8. JMS-backed Message Channels . . . 559
24.9. Using JMS Message Selectors . . . 561
24.10. JMS Samples . . . 561
25. Mail Support . . . 563
25.1. Mail-sending Channel Adapter . . . 563
25.2. Mail-receiving Channel Adapter . . . 564
25.3. Inbound Mail Message Mapping . . . 565
25.4. Mail Namespace Support . . . 567
25.5. Marking IMAP Messages When `\Recent` Is Not Supported . . . 573
25.6. Email Message Filtering . . . 573
25.7. Transaction Synchronization . . . 574
25.8. Configuring channel adapters with the Java DSL . . . 576
26. MongoDb Support . . . 577
26.1. Connecting to MongoDb . . . 577

26.2. MongoDB Message Store . . . 579
26.3. MongoDB Inbound Channel Adapter . . . 582
26.4. MongoDB Change Stream Inbound Channel Adapter . . . 584
26.5. MongoDB Outbound Channel Adapter . . . 585
26.6. MongoDB Outbound Gateway . . . 586
26.7. MongoDB Reactive Channel Adapters . . . 589
27. MQTT Support . . . 591
27.1. Inbound (Message-driven) Channel Adapter . . . 591
27.2. Outbound Channel Adapter . . . 597
27.3. Events . . . 601
28. Redis Support . . . 602
28.1. Connecting to Redis . . . 602
28.2. Messaging with Redis . . . 604
28.3. Redis Message Store . . . 610
28.4. Redis Metadata Store . . . 611
28.5. Redis Store Inbound Channel Adapter . . . 612
28.6. RedisStore Outbound Channel Adapter . . . 615
28.7. Redis Outbound Command Gateway . . . 616
28.8. Redis Queue Outbound Gateway . . . 618
28.9. Redis Queue Inbound Gateway . . . 619
28.10. Redis Lock Registry . . . 621
29. Resource Support . . . 622
29.1. Resource Inbound Channel Adapter . . . 622
30. RMI Support . . . 624
30.1. Outbound RMI . . . 624
30.2. Inbound RMI . . . 624
30.3. RMI namespace support . . . 625
30.4. Configuring with Java Configuration . . . 626
31. RSocket Support . . . 627
31.1. RSocket Inbound Gateway . . . 629
31.2. RSocket Outbound Gateway . . . 630
31.3. RSocket Namespace Support . . . 631
31.4. Configuring RSocket Endpoints with Java . . . 633
32. SFTP Adapters . . . 636
32.1. SFTP Session Factory . . . 636
32.2. Proxy Factory Bean . . . 639
32.3. Delegating Session Factory . . . 639
32.4. SFTP Session Caching . . . 640
32.5. Using `RemoteFileTemplate` . . . 641
32.6. SFTP Inbound Channel Adapter . . . 642
32.7. SFTP Streaming Inbound Channel Adapter . . . 649

32.8. Inbound Channel Adapters: Polling Multiple Servers and Directories . . . 652
32.9. Inbound Channel Adapters: Controlling Remote File Fetching . . . 653
32.10. SFTP Outbound Channel Adapter . . . 654
32.11. SFTP Outbound Gateway . . . 659
32.12. SFTP/JSCH Logging . . . 667
32.13. MessageSessionCallback . . . 668
32.14. Apache Mina SFTP Server Events . . . 668
32.15. Remote File Information . . . 669
33. STOMP Support . . . 671
33.1. Overview . . . 671
33.2. STOMP Inbound Channel Adapter . . . 672
33.3. STOMP Outbound Channel Adapter . . . 672
33.4. STOMP Headers Mapping . . . 672
33.5. STOMP Integration Events . . . 673
33.6. STOMP Adapters Java Configuration . . . 673
33.7. STOMP Namespace Support . . . 675
34. Stream Support . . . 678
34.1. Reading from Streams . . . 678
34.2. Writing to Streams . . . 680
34.3. Stream Namespace Support . . . 680
35. Syslog Support . . . 682
35.1. Syslog Inbound Channel Adapter . . . 682
36. TCP and UDP Support . . . 686
36.1. Introduction . . . 686
36.2. UDP Adapters . . . 687
36.3. TCP Connection Factories . . . 691
36.4. Testing Connections . . . 699
36.5. TCP Connection Interceptors . . . 700
36.6. TCP Connection Events . . . 701
36.7. TCP Adapters . . . 702
36.8. TCP Gateways . . . 704
36.9. TCP Message Correlation . . . 706
36.10. About Non-blocking I/O (NIO) . . . 709
36.11. SSL/TLS Support . . . 713
36.12. Advanced Techniques . . . 715
36.13. IP Configuration Attributes . . . 720
36.14. IP Message Headers . . . 727
36.15. Annotation-Based Configuration . . . 728
37. WebFlux Support . . . 731
37.1. WebFlux Inbound Components . . . 731
37.2. WebFlux Outbound Components . . . 733

37.3. WebFlux Namespace Support . . . 734
37.4. Configuring WebFlux Endpoints with Java . . . 737
37.5. WebFlux Header Mappings . . . 739
38. WebSockets Support . . . 740
38.1. Overview . . . 741
38.2. WebSocket Inbound Channel Adapter . . . 742
38.3. WebSocket Outbound Channel Adapter . . . 743
38.4. WebSockets Namespace Support . . . 743
38.5. Using `ClientStompEncoder` . . . 749
39. Web Services Support . . . 751
39.1. Outbound Web Service Gateways . . . 751
39.2. Inbound Web Service Gateways . . . 752
39.3. Web Service Namespace Support . . . 753
39.4. Web Service Java DSL Support . . . 754
39.5. Outbound URI Configuration . . . 756
39.6. WS Message Headers . . . 757
39.7. MTOM Support . . . 760
40. XML Support - Dealing with XML Payloads . . . 762
40.1. Namespace Support . . . 763
40.2. Transforming XML Payloads . . . 767
40.3. Transforming XML Messages with XPath . . . 774
40.4. Splitting XML Messages . . . 776
40.5. Routing XML Messages with XPath . . . 778
40.6. XPath Header Enricher . . . 780
40.7. Using the XPath Filter . . . 783
40.8. #xpath SpEL Function . . . 784
40.9. XML Validating Filter . . . 785
41. XMPP Support . . . 787
41.1. XMPP Connection . . . 788
41.2. XMPP Messages . . . 788
41.3. XMPP Presence . . . 791
41.4. Advanced Configuration . . . 792
41.5. XMPP Message Headers . . . 794
41.6. XMPP Extensions . . . 794
42. Zookeeper Support . . . 798
42.1. Zookeeper Metadata Store . . . 798
42.2. Zookeeper Lock Registry . . . 799
42.3. Zookeeper Leadership Event Handling . . . 799
Appendices . . . 801
Appendix A: Error Handling . . . 802
Appendix B: Spring Expression Language (SpEL) . . . 804

B.1. SpEL Evaluation Context Customization . . . 804
B.2. SpEL Functions . . . 806
B.3. Property Accessors . . . 808
Appendix C: Message Publishing . . . 809
C.1. Message Publishing Configuration . . . 809
Appendix D: Transaction Support . . . 818
D.1. Understanding Transactions in Message flows . . . 818
D.2. Transaction Boundaries . . . 821
D.3. Transaction Synchronization . . . 821
D.4. Pseudo Transactions . . . 824
D.5. Reactive Transactions . . . 824
Appendix E: Security in Spring Integration . . . 825
E.1. Securing channels . . . 825
E.2. Security Context Propagation . . . 827
Appendix F: Configuration . . . 830
F.1. Namespace Support . . . 830
F.2. Configuring the Task Scheduler . . . 832
F.3. Global Properties . . . 833
F.4. Annotation Support . . . 835
F.5. Messaging Meta-Annotations . . . 842
F.6. Message Mapping Rules and Conventions . . . 847
Appendix G: Testing support . . . 853
G.1. Testing Utilities . . . 854
G.2. Spring Integration and the Test Context . . . 858
G.3. Integration Mocks . . . 860
G.4. Other Resources . . . 862
Appendix H: Spring Integration Samples . . . 863
H.1. Where to Get Samples . . . 863
H.2. Submitting Samples or Sample Requests . . . 864
H.3. Samples Structure . . . 864
H.4. Samples . . . 866
Appendix I: Additional Resources . . . 878
Appendix J: Change History . . . 879
J.1. Changes between 5.1 and 5.2 . . . 879
J.2. Package and Class Changes . . . 879
J.3. Behavior Changes . . . 879
J.4. New Components . . . 879
J.5. General Changes . . . 880
J.6. Changes between 5.0 and 5.1 . . . 882
J.7. Changes between 4.3 and 5.0 . . . 887
J.8. Changes between 4.2 and 4.3 . . . 894

J.9. Changes between 4.1 and 4.2 900
J.10. Changes between 4.0 and 4.1 907
J.11. Changes between 3.0 and 4.0 911
J.12. Changes Between 2.2 and 3.0 916
J.13. Changes between 2.1 and 2.2 925
J.14. Changes between 2.0 and 2.1 928
J.15. Changes between Versions 1.0 and 2.0 933

© 2009 - 2020 Pivotal Software, Inc. All rights reserved.

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Preface

This chapter includes:

- Requirements
- Code Conventions
- Conventions in This Guide

Chapter 1. Requirements

This section details the compatible Java and Spring Framework versions.

1.1. Compatible Java Versions

For Spring Integration 5.2.x, the minimum compatible Java version is Java SE 8. Older versions of Java are not supported.

1.2. Compatible Versions of the Spring Framework

Spring Integration 5.2.x requires Spring Framework 5.2 or later.

Chapter 2. Code Conventions

Spring Framework 2.0 introduced support for namespaces, which simplifies the XML configuration of the application context and lets Spring Integration provide broad namespace support.

In this reference guide, the `int` namespace prefix is used for Spring Integration's core namespace support. Each Spring Integration adapter type (also called a module) provides its own namespace, which is configured by using the following convention:

The following example shows the `int`, `int-event`, and `int-stream` namespaces in use:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-webflux="http://www.springframework.org/schema/integration/webflux"
  xmlns:int-stream="http://www.springframework.org/schema/integration/stream"
  xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   https://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/integration
   https://www.springframework.org/schema/integration/spring-integration.xsd
   http://www.springframework.org/schema/integration/webflux
   https://www.springframework.org/schema/integration/webflux/spring-integration-
webflux.xsd
   http://www.springframework.org/schema/integration/stream
   https://www.springframework.org/schema/integration/stream/spring-integration-
stream.xsd">
...
</beans>
```

For a detailed explanation regarding Spring Integration's namespace support, see Namespace Support.

The namespace prefix can be freely chosen. You may even choose not to use any namespace prefixes at all. Therefore, you should apply the convention that best suits your application. Be aware, though, that SpringSource Tool Suite™ (STS) uses the same namespace conventions for Spring Integration as used in this reference guide.

Chapter 3. Conventions in This Guide

In some cases, to aid formatting when specifying long fully qualified class names, we shorten `org.springframework` to `o.s` and `org.springframework.integration` to `o.s.i`, such as with `o.s.i.transaction.TransactionSynchronizationFactory`.

What's New?

For those who are already familiar with Spring Integration, this chapter provides a brief overview of the new features of version 5.3.

If you are interested in the changes and features that were introduced in earlier versions, see the Change History.

Chapter 4. What's New in Spring Integration 5.3?

If you are interested in more details, see the Issue Tracker tickets that were resolved as part of the 5.3 development process.

4.1. New Components

4.1.1. Integration Pattern

The `IntegrationPattern` abstraction has been introduced to indicate which enterprise integration pattern (an `IntegrationPatternType`) and category a Spring Integration component belongs to. See its JavaDocs and Integration Graph for more information about this abstraction and its use-cases.

4.1.2. `ReactiveMessageHandler`

The `ReactiveMessageHandler` is now natively supported in the framework. See ReactiveMessageHandler for more information.

4.1.3. `ReactiveMessageSourceProducer`

The `ReactiveMessageSourceProducer` is a reactive implementation of the `MessageProducerSupport` to wrap a provided `MessageSource` into a `Flux` for on demand `receive()` calls. See Reactive Streams Support for more information.

4.1.4. Java DSL Extensions

A new `IntegrationFlowExtension` API has been introduced to allow extension of the existing Java DSL with custom or composed EIP-operators. This also can be used to introduce customizers for any out-of-the-box `IntegrationComponentSpec` extensions. See DSL Extensions for more information.

4.1.5. Kotlin DSL

The Kotlin DSL for integration flow configurations has been introduced. See Kotlin DSL Chapter for more information.

4.1.6. ReactiveRequestHandlerAdvice

A `ReactiveRequestHandlerAdvice` is provided to customize `Mono` replies from message handlers. See Reactive Advice for more information.

4.1.7. HandleMessageAdviceAdapter

A `HandleMessageAdviceAdapter` is provided to wrap any `MethodInterceptor` for applying on the `MessageHandler.handleMessage()` instead of a default `AbstractReplyProducingMessageHandler.RequestHandler.handleRequestMessage()` behavior. See

Handling Message Advice for more information.

4.1.8. MongoDB Reactive Channel Adapters

The `spring-integration-mongodb` module now provides channel adapter implementations for the Reactive MongoDb driver support in Spring Data. Also, a reactive implementation for MongoDb change stream support is present with the `MongoDbChangeStreamMessageProducer`. See MongoDB Support for more information.

4.1.9. ReceiveMessageAdvice

A special `ReceiveMessageAdvice` has been introduced to proxy exactly `MessageSource.receive()` or `PollableChannel.receive()`. See Smart Polling for more information.

4.2. General Changes

The gateway proxy now doesn't proxy `default` methods by default. See Invoking `default` Methods for more information.

Internal components (such as `_org.springframework.integration.errorLogger`) now have a shortened name when they are represented in the integration graph. See Integration Graph for more information.

In the aggregator, when the `MessageGroupProcessor` returns a `Message`, the `MessageBuilder.popSequenceDetails()` is performed on the output message if the `sequenceDetails` matches the header in the first message of the group. See Aggregator Programming Model for more information.

A new `publishSubscribeChannel()` operator, based on the `BroadcastCapableChannel` and `BroadcastPublishSubscribeSpec`, was added into Java DSL. This fluent API has its advantage when we configure sub-flows as pub-sub subscribers for broker-backed channels like `SubscribableJmsChannel`, `SubscribableRedisChannel` etc. See Sub-flows support for more information.

Transactional support in Spring Integration now also includes options to configure a `ReactiveTransactionManager` if a `MessageSource` or `MessageHandler` implementation produces a reactive type for payload to send. See `TransactionInterceptorBuilder` for more information. See also Reactive Transactions.

A new `intercept()` operator to register `ChannelInterceptor` instances without creating explicit channels was added into Java DSL. See Operator intercept() for more information.

The `MessageStoreSelector` has a new mechanism to compare an old and new value. See Idempotent Receiver Enterprise Integration Pattern for more information.

The `MessageProducerSupport` base class now has a `subscribeToPublisher(Publisher<? extends Message<?>>)` API to allow implementation of message-driven producer endpoints which emit messages via reactive `Publisher`. See Reactive Streams Support for more information.

4.3. AMQP Changes

The outbound channel adapter has a new property `multiSend` allowing multiple messages to be sent within the scope of one `RabbitTemplate` invocation. See AMQP Outbound Channel Adapter for more information.

The inbound channel adapter now supports a listener container with the `consumerBatchEnabled` property set to `true`. See AMQP Inbound Channel Adapter

4.4. HTTP Changes

The `encodeUri` property on the `AbstractHttpRequestExecutingMessageHandler` has been deprecated in favor of newly introduced `encodingMode`. See `DefaultUriBuilderFactory.EncodingMode` JavaDocs and Controlling URI Encoding for more information. This also affects `WebFluxRequestExecutingMessageHandler`, respective Java DSL and XML configuration. The same option is added into an `AbstractWebServiceOutboundGateway`.

4.5. Web Services Changes

Java DSL support has been added for Web Service components. The `encodeUri` property on the `AbstractWebServiceOutboundGateway` has been deprecated in favor of newly introduced `encodingMode` - similar to HTTP changes above. See Web Services Support for more information.

4.6. TCP Changes

The `FailoverClientConnectionFactory` no longer fails back, by default, until the current connection fails. See TCP Failover Client Connection Factory for more information.

The `TcpOutboundGateway` now supports asynchronous request/reply. See TCP Gateways for more information.

You can now configure client connections to perform some arbitrary test on new connections. See Testing Connections for more information.

4.7. RSocket Changes

A `decodeFluxAsUnit` option has been added to the `RSocketInboundGateway` with the meaning to decode incoming `Flux` as a single unit or apply decoding for each event in it. See RSocket Inbound Gateway for more information.

4.8. Zookeeper Changes

A `LeaderInitiatorFactoryBean` (as well as its XML `<int-zk:leader-listener>`) exposes a `candidate` option for more control over a `Candidate` configuration. See Leadership event handling for more information.

4.9. MQTT Changes

The inbound channel adapter can now be configured to provide user control over when a message is acknowledged as being delivered. See Manual Acks for more information.

The outbound adapter now publishes a `MqttConnectionFailedEvent` when a connection can't be created, or is lost. Previously, only the inbound adapter did so. See MQTT Events.

4.10. (S)FTP Changes

The `FileTransferringMessageHandler` (for FTP and SFTP, for example) in addition to `File`, `byte[]`, `String` and `InputStream` now also supports an `org.springframework.core.io.Resource`. See SFTP Support and FTP Support for more information.

4.11. File Changes

The `FileSplitter` doesn't require a Jackson processor (or similar) dependency any more for the `markersJson` mode. It uses a `SimpleJsonSerializer` for a straightforward string representation of the `FileSplitter.FileMarker` instances. See FileSplitter for more information.

Overview of Spring Integration Framework

Spring Integration provides an extension of the Spring programming model to support the well known Enterprise Integration Patterns. It enables lightweight messaging within Spring-based applications and supports integration with external systems through declarative adapters. Those adapters provide a higher level of abstraction over Spring's support for remoting, messaging, and scheduling.

Spring Integration's primary goal is to provide a simple model for building enterprise integration solutions while maintaining the separation of concerns that is essential for producing maintainable, testable code.

Chapter 5. Spring Integration Overview

This chapter provides a high-level introduction to Spring Integration's core concepts and components. It includes some programming tips to help you make the most of Spring Integration.

5.1. Background

One of the key themes of the Spring Framework is Inversion of Control (IoC). In its broadest sense, this means that the framework handles responsibilities on behalf of the components that are managed within its context. The components themselves are simplified, because they are relieved of those responsibilities. For example, dependency injection relieves the components of the responsibility of locating or creating their dependencies. Likewise, aspect-oriented programming relieves business components of generic cross-cutting concerns by modularizing them into reusable aspects. In each case, the end result is a system that is easier to test, understand, maintain, and extend.

Furthermore, the Spring framework and portfolio provide a comprehensive programming model for building enterprise applications. Developers benefit from the consistency of this model and especially from the fact that it is based upon well established best practices, such as programming to interfaces and favoring composition over inheritance. Spring's simplified abstractions and powerful support libraries boost developer productivity while simultaneously increasing the level of testability and portability.

Spring Integration is motivated by these same goals and principles. It extends the Spring programming model into the messaging domain and builds upon Spring's existing enterprise integration support to provide an even higher level of abstraction. It supports message-driven architectures where inversion of control applies to runtime concerns, such as when certain business logic should run and where the response should be sent. It supports routing and transformation of messages so that different transports and different data formats can be integrated without impacting testability. In other words, the messaging and integration concerns are handled by the framework. Business components are further isolated from the infrastructure, and developers are relieved of complex integration responsibilities.

As an extension of the Spring programming model, Spring Integration provides a wide variety of configuration options, including annotations, XML with namespace support, XML with generic "bean" elements, and direct usage of the underlying API. That API is based upon well defined strategy interfaces and non-invasive, delegating adapters. Spring Integration's design is inspired by the recognition of a strong affinity between common patterns within Spring and the well known patterns described in *Enterprise Integration Patterns*, by Gregor Hohpe and Bobby Woolf (Addison Wesley, 2004). Developers who have read that book should be immediately comfortable with the Spring Integration concepts and terminology.

5.2. Goals and Principles

Spring Integration is motivated by the following goals:

- Provide a simple model for implementing complex enterprise integration solutions.

- Facilitate asynchronous, message-driven behavior within a Spring-based application.
- Promote intuitive, incremental adoption for existing Spring users.

Spring Integration is guided by the following principles:

- Components should be loosely coupled for modularity and testability.
- The framework should enforce separation of concerns between business logic and integration logic.
- Extension points should be abstract in nature (but within well-defined boundaries) to promote reuse and portability.

5.3. Main Components

From a vertical perspective, a layered architecture facilitates separation of concerns, and interface-based contracts between layers promote loose coupling. Spring-based applications are typically designed this way, and the Spring framework and portfolio provide a strong foundation for following this best practice for the full stack of an enterprise application. Message-driven architectures add a horizontal perspective, yet these same goals are still relevant. Just as “layered architecture” is an extremely generic and abstract paradigm, messaging systems typically follow the similarly abstract “pipes-and-filters” model. The “filters” represent any components capable of producing or consuming messages, and the “pipes” transport the messages between filters so that the components themselves remain loosely-coupled. It is important to note that these two high-level paradigms are not mutually exclusive. The underlying messaging infrastructure that supports the “pipes” should still be encapsulated in a layer whose contracts are defined as interfaces. Likewise, the “filters” themselves should be managed within a layer that is logically above the application’s service layer, interacting with those services through interfaces in much the same way that a web tier would.

5.3.1. Message

In Spring Integration, a message is a generic wrapper for any Java object combined with metadata used by the framework while handling that object. It consists of a payload and headers. The payload can be of any type, and the headers hold commonly required information such as ID, timestamp, correlation ID, and return address. Headers are also used for passing values to and from connected transports. For example, when creating a message from a received file, the file name may be stored in a header to be accessed by downstream components. Likewise, if a message’s content is ultimately going to be sent by an outbound mail adapter, the various properties (to, from, cc, subject, and others) may be configured as message header values by an upstream component. Developers can also store any arbitrary key-value pairs in the headers.

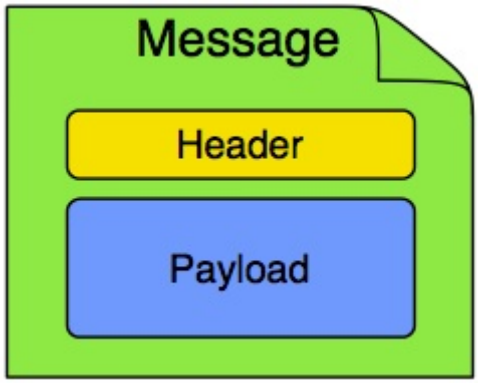


Figure 1. Message

5.3.2. Message Channel

A message channel represents the “pipe” of a pipes-and-filters architecture. Producers send messages to a channel, and consumers receive messages from a channel. The message channel therefore decouples the messaging components and also provides a convenient point for interception and monitoring of messages.

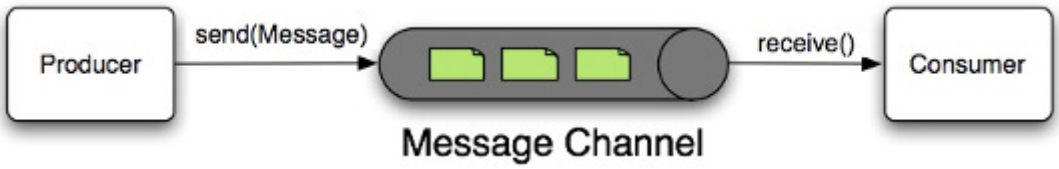


Figure 2. Message Channel

A message channel may follow either point-to-point or publish-subscribe semantics. With a point-to-point channel, no more than one consumer can receive each message sent to the channel. Publish-subscribe channels, on the other hand, attempt to broadcast each message to all subscribers on the channel. Spring Integration supports both of these models.

Whereas “point-to-point” and "publish-subscribe" define the two options for how many consumers ultimately receive each message, there is another important consideration: Should the channel buffer messages? In Spring Integration, pollable channels are capable of buffering Messages within a queue. The advantage of buffering is that it allows for throttling the inbound messages and thereby prevents overloading a consumer. However, as the name suggests, this also adds some complexity, since a consumer can only receive the messages from such a channel if a poller is configured. On the other hand, a consumer connected to a subscribable channel is simply message-driven. Message Channel Implementations has a detailed discussion of the variety of channel implementations available in Spring Integration.

5.3.3. Message Endpoint

One of the primary goals of Spring Integration is to simplify the development of enterprise integration solutions through inversion of control. This means that you should not have to implement consumers and producers directly, and you should not even have to build messages and invoke send or receive operations on a message channel. Instead, you should be able to focus on your specific domain model with an implementation based on plain objects. Then, by providing declarative configuration, you can “connect” your domain-specific code to the messaging infrastructure provided by Spring Integration. The components responsible for these connections

are message endpoints. This does not mean that you should necessarily connect your existing application code directly. Any real-world enterprise integration solution requires some amount of code focused upon integration concerns such as routing and transformation. The important thing is to achieve separation of concerns between the integration logic and the business logic. In other words, as with the Model-View-Controller (MVC) paradigm for web applications, the goal should be to provide a thin but dedicated layer that translates inbound requests into service layer invocations and then translates service layer return values into outbound replies. The next section provides an overview of the message endpoint types that handle these responsibilities, and, in upcoming chapters, you can see how Spring Integration's declarative configuration options provide a non-invasive way to use each of these.

5.4. Message Endpoints

A Message Endpoint represents the "filter" of a pipes-and-filters architecture. As mentioned earlier, the endpoint's primary role is to connect application code to the messaging framework and to do so in a non-invasive manner. In other words, the application code should ideally have no awareness of the message objects or the message channels. This is similar to the role of a controller in the MVC paradigm. Just as a controller handles HTTP requests, the message endpoint handles messages. Just as controllers are mapped to URL patterns, message endpoints are mapped to message channels. The goal is the same in both cases: isolate application code from the infrastructure. These concepts and all of the patterns that follow are discussed at length in the *Enterprise Integration Patterns* book. Here, we provide only a high-level description of the main endpoint types supported by Spring Integration and the roles associated with those types. The chapters that follow elaborate and provide sample code as well as configuration examples.

5.4.1. Message Transformer

A message transformer is responsible for converting a message's content or structure and returning the modified message. Probably the most common type of transformer is one that converts the payload of the message from one format to another (such as from XML to `java.lang.String`). Similarly, a transformer can add, remove, or modify the message's header values.

5.4.2. Message Filter

A message filter determines whether a message should be passed to an output channel at all. This simply requires a boolean test method that may check for a particular payload content type, a property value, the presence of a header, or other conditions. If the message is accepted, it is sent to the output channel. If not, it is dropped (or, for a more severe implementation, an `Exception` could be thrown). Message filters are often used in conjunction with a publish-subscribe channel, where multiple consumers may receive the same message and use the criteria of the filter to narrow down the set of messages to be processed.

Be careful not to confuse the generic use of “filter” within the pipes-and-filters architectural pattern with this specific endpoint type that selectively narrows down the messages flowing between two channels. The pipes-and-filters concept of a “filter” matches more closely with Spring Integration’s message endpoint: any component that can be connected to a message channel in order to send or receive messages.

5.4.3. Message Router

A message router is responsible for deciding what channel or channels (if any) should receive the message next. Typically, the decision is based upon the message’s content or the metadata available in the message headers. A message router is often used as a dynamic alternative to a statically configured output channel on a service activator or other endpoint capable of sending reply messages. Likewise, a message router provides a proactive alternative to the reactive message filters used by multiple subscribers, as described earlier.

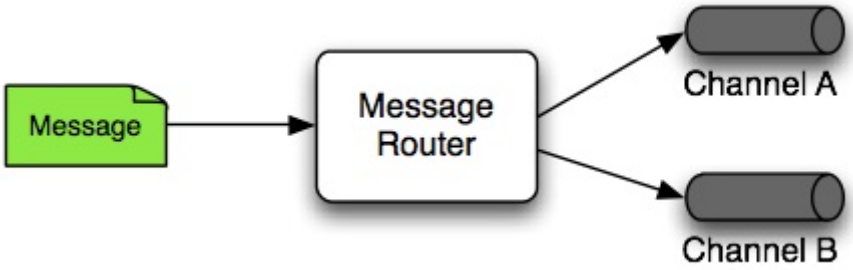


Figure 3. Message Router

5.4.4. Splitter

A splitter is another type of message endpoint whose responsibility is to accept a message from its input channel, split that message into multiple messages, and send each of those to its output channel. This is typically used for dividing a “composite” payload object into a group of messages containing the subdivided payloads.

5.4.5. Aggregator

Basically a mirror-image of the splitter, the aggregator is a type of message endpoint that receives multiple messages and combines them into a single message. In fact, aggregators are often downstream consumers in a pipeline that includes a splitter. Technically, the aggregator is more complex than a splitter, because it is required to maintain state (the messages to be aggregated), to decide when the complete group of messages is available, and to timeout if necessary. Furthermore, in case of a timeout, the aggregator needs to know whether to send the partial results, discard them, or send them to a separate channel. Spring Integration provides a `CorrelationStrategy`, a `ReleaseStrategy`, and configurable settings for timeout, whether to send partial results upon timeout, and a discard channel.

5.4.6. Service Activator

A Service Activator is a generic endpoint for connecting a service instance to the messaging system. The input message channel must be configured, and, if the service method to be invoked is capable

of returning a value, an output message Channel may also be provided.

The output channel is optional, since each message may also provide its own 'Return Address' header. This same rule applies for all consumer endpoints.

The service activator invokes an operation on some service object to process the request message, extracting the request message's payload and converting (if the method does not expect a message-typed parameter). Whenever the service object's method returns a value, that return value is likewise converted to a reply message if necessary (if it is not already a message type). That reply message is sent to the output channel. If no output channel has been configured, the reply is sent to the channel specified in the message's "return address", if available.

A request-reply service activator endpoint connects a target object's method to input and output Message Channels.

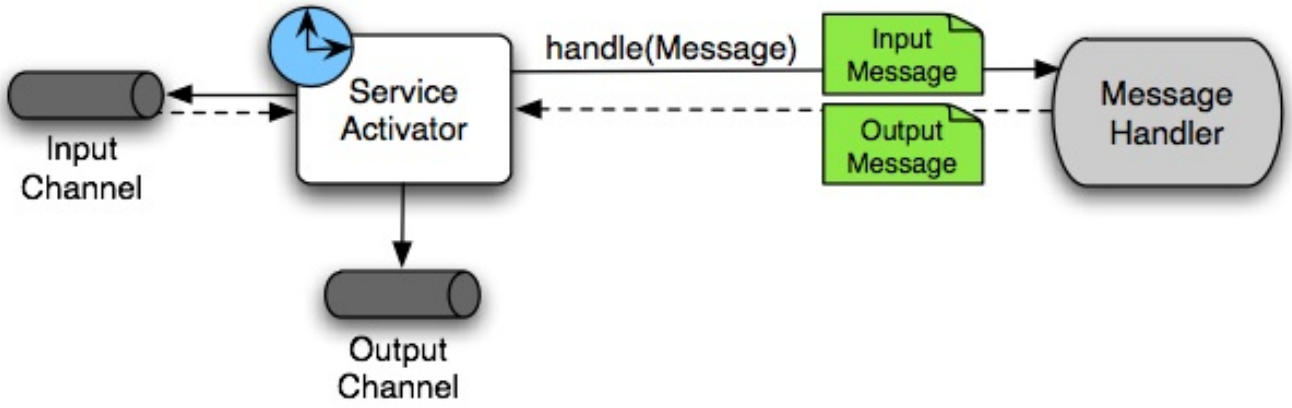


Figure 4. Service Activator

As discussed earlier, in Message Channel, channels can be pollable or subscribable. In the preceding diagram, this is depicted by the "clock" symbol and the solid arrow (poll) and the dotted arrow (subscribe).

5.4.7. Channel Adapter

A channel adapter is an endpoint that connects a message channel to some other system or transport. Channel adapters may be either inbound or outbound. Typically, the channel adapter does some mapping between the message and whatever object or resource is received from or sent to the other system (file, HTTP Request, JMS message, and others). Depending on the transport, the channel adapter may also populate or extract message header values. Spring Integration provides a number of channel adapters, which are described in upcoming chapters.

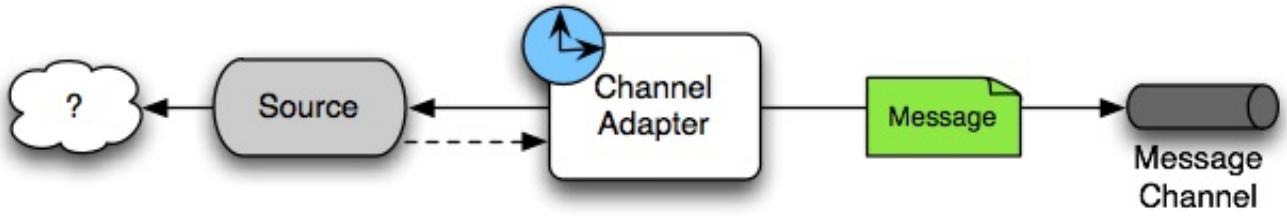
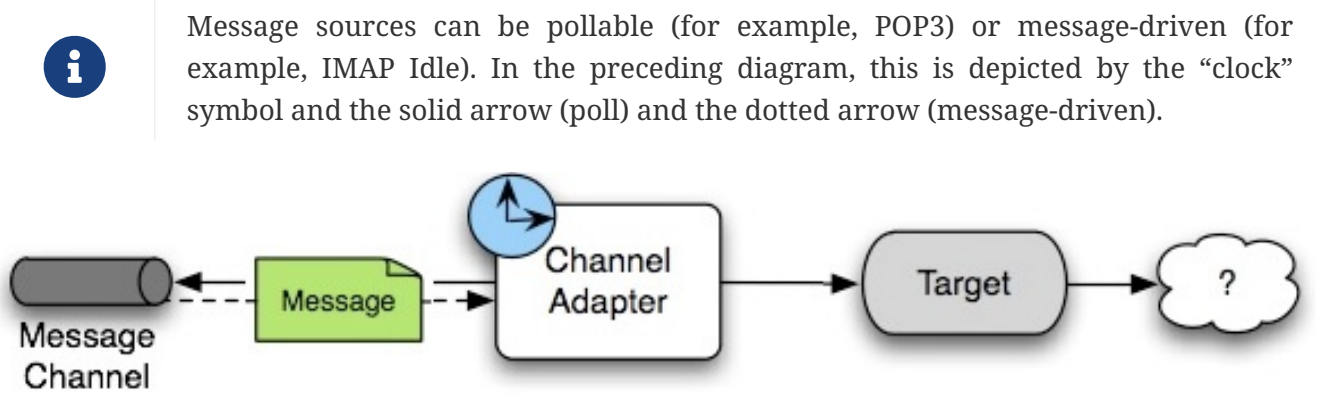
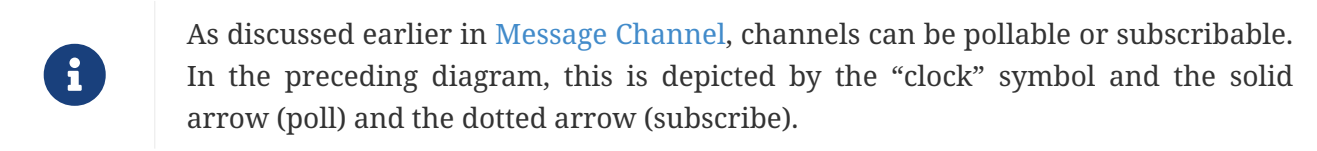


Figure 5. An inbound channel adapter endpoint connects a source system to a `MessageChannel`.



Message sources can be pollable (for example, POP3) or message-driven (for example, IMAP Idle). In the preceding diagram, this is depicted by the “clock” symbol and the solid arrow (poll) and the dotted arrow (message-driven).

Figure 6. An outbound channel adapter endpoint connects a `MessageChannel` to a target system.



As discussed earlier in Message Channel, channels can be pollable or subscribable. In the preceding diagram, this is depicted by the “clock” symbol and the solid arrow (poll) and the dotted arrow (subscribe).

5.4.8. Endpoint Bean Names

Consuming endpoints (anything with an `inputChannel`) consist of two beans, the consumer and the message handler. The consumer has a reference to the message handler and invokes it as messages arrive.

Consider the following XML example:

```
<int:service-activator id = "someService" ... />
```

Given the preceding example, the bean names are as follows:

- Consumer: `someService` (the `id`)
- Handler: `someService.handler`

When using Enterprise Integration Pattern (EIP) annotations, the names depend on several factors. Consider the following example of an annotated POJO:

```
@Component
public class SomeComponent {

    @ServiceActivator(inputChannel = ...)
    public String someMethod(...) {
        ...
    }

}
```

Given the preceding example, the bean names are as follows:

- Consumer: `someComponent.someMethod.serviceActivator`
- Handler: `someComponent.someMethod.serviceActivator.handler`

Starting with version 5.0.4, you can modify these names by using the `@EndpointId` annotation, as the following example shows:

```
@Component
public class SomeComponent {

    @EndpointId("someService")
    @ServiceActivator(inputChannel = ...)
    public String someMethod(...) {
        ...
    }

}
```

Given the preceding example, the bean names are as follows:

- Consumer: `someService`
- Handler: `someService.handler`

The `@EndpointId` creates names as created by the `id` attribute with XML configuration. Consider the following example of an annotated bean:

```
@Configuration
public class SomeConfiguration {

    @Bean
    @ServiceActivator(inputChannel = ...)
    public MessageHandler someHandler() {
        ...
    }

}
```

Given the preceding example, the bean names are as follows:

- Consumer: `someConfiguration.someHandler.serviceActivator`
- Handler: `someHandler` (the `@Bean` name)

Starting with version 5.0.4, you can modify these names by using the `@EndpointId` annotation, as the

following example shows:

```
@Configuration
public class SomeConfiguration {

    @Bean("someService.handler")             ①
    @EndpointId("someService")               ②
    @ServiceActivator(inputChannel = ...)
    public MessageHandler someHandler() {
        ...
    }

}
```

① Handler: `someService.handler` (the bean name)

② Consumer: `someService` (the endpoint ID)

The `@EndpointId` annotation creates names as created by the `id` attribute with XML configuration, as long as you use the convention of appending `.handler` to the `@Bean` name.

There is one special case where a third bean is created: For architectural reasons, if a `MessageHandler` `@Bean` does not define an `AbstractReplyProducingMessageHandler`, the framework wraps the provided bean in a `ReplyProducingMessageHandlerWrapper`. This wrapper supports request handler advice handling and emits the normal 'produced no reply' debug log messages. Its bean name is the handler bean name plus `.wrapper` (when there is an `@EndpointId` — otherwise, it is the normal generated handler name).

Similarly Pollable Message Sources create two beans, a `SourcePollingChannelAdapter` (SPCA) and a `MessageSource`.

Consider the following XML configuration:

```
<int:inbound-channel-adapter id = "someAdapter" ... />
```

Given the preceding XML configuration, the bean names are as follows:

- SPCA: `someAdapter` (the `id`)
- Handler: `someAdapter.source`

Consider the following Java configuration of a POJO to define an `@EndpointId`:

```
@EndpointId("someAdapter")
@InboundChannelAdapter(channel = "channel3", poller = @Poller(fixedDelay = "5000"
))
public String pojoSource() {
    ...
}
```

Given the preceding Java configuration example, the bean names are as follows:

- SPCA: `someAdapter`
- Handler: `someAdapter.source`

Consider the following Java configuration of a bean to define an `@EndpointID`:

```
@Bean("someAdapter.source")
@EndpointId("someAdapter")
@InboundChannelAdapter(channel = "channel3", poller = @Poller(fixedDelay = "5000"
))
public MessageSource<?> source() {
    return () -> {
        ...
    };
}
```

Given the preceding example, the bean names are as follows:

- SPCA: `someAdapter`
- Handler: `someAdapter.source` (as long as you use the convention of appending `.source` to the `@Bean` name)

5.5. Configuration and `@EnableIntegration`

Throughout this document, you can see references to XML namespace support for declaring elements in a Spring Integration flow. This support is provided by a series of namespace parsers that generate appropriate bean definitions to implement a particular component. For example, many endpoints consist of a `MessageHandler` bean and a `ConsumerEndpointFactoryBean` into which the handler and an input channel name are injected.

The first time a Spring Integration namespace element is encountered, the framework automatically declares a number of beans (a task scheduler, an implicit channel creator, and others) that are used to support the runtime environment.

Version 4.0 introduced the `@EnableIntegration` annotation, to allow the registration of Spring Integration infrastructure beans (see the Javadoc). This annotation is required when only Java configuration is used — for example with Spring Boot or Spring Integration Messaging Annotation support and Spring Integration Java DSL with no XML integration configuration.

The `@EnableIntegration` annotation is also useful when you have a parent context with no Spring Integration components and two or more child contexts that use Spring Integration. It lets these common components be declared once only, in the parent context.

The `@EnableIntegration` annotation registers many infrastructure components with the application context. In particular, it:

- Registers some built-in beans, such as `errorChannel` and its `LoggingHandler`, `taskScheduler` for pollers, `jsonPath` SpEL-function, and others.
- Adds several `BeanFactoryPostProcessor` instances to enhance the `BeanFactory` for global and default integration environment.
- Adds several `BeanPostProcessor` instances to enhance or convert and wrap particular beans for integration purposes.
- Adds annotation processors to parse messaging annotations and registers components for them with the application context.

The `@IntegrationComponentScan` annotation also permits classpath scanning. This annotation plays a similar role as the standard Spring Framework `@ComponentScan` annotation, but it is restricted to components and annotations that are specific to Spring Integration, which the standard Spring Framework component scan mechanism cannot reach. For an example, see `@MessagingGateway` Annotation.

The `@EnablePublisher` annotation registers a `PublisherAnnotationBeanPostProcessor` bean and configures the `default-publisher-channel` for those `@Publisher` annotations that are provided without a `channel` attribute. If more than one `@EnablePublisher` annotation is found, they must all have the same value for the default channel. See Annotation-driven Configuration with the `@Publisher` Annotation for more information.

The `@GlobalChannelInterceptor` annotation has been introduced to mark `ChannelInterceptor` beans for global channel interception. This annotation is an analogue of the `<int:channel-interceptor>` XML element (see Global Channel Interceptor Configuration). `@GlobalChannelInterceptor` annotations can be placed at the class level (with a `@Component` stereotype annotation) or on `@Bean` methods within `@Configuration` classes. In either case, the bean must implement `ChannelInterceptor`.

Starting with version 5.1, global channel interceptors apply to dynamically registered channels — such as beans that are initialized by using `beanFactory.initializeBean()` or through the `IntegrationFlowContext` when using the Java DSL. Previously, interceptors were not applied when beans were created after the application context was refreshed.

The `@IntegrationConverter` annotation marks `Converter`, `GenericConverter`, or `ConverterFactory` beans as candidate converters for `integrationConversionService`. This annotation is an analogue of the `<int:converter>` XML element (see Payload Type Conversion). You can place `@IntegrationConverter`

annotations at the class level (with a `@Component` stereotype annotation) or on `@Bean` methods within `@Configuration` classes.

See Annotation Support for more information about messaging annotations.

5.6. Programming Considerations

You should use plain old java objects (POJOs) whenever possible and only expose the framework in your code when absolutely necessary. See POJO Method invocation for more information.

If you do expose the framework to your classes, there are some considerations that need to be taken into account, especially during application startup:

- If your component is `ApplicationContextAware`, you should generally not use the `ApplicationContext` in the `setApplicationContext()` method. Instead, store a reference and defer such uses until later in the context lifecycle.
- If your component is an `InitializingBean` or uses `@PostConstruct` methods, do not send any messages from these initialization methods. The application context is not yet initialized when these methods are called, and sending such messages is likely to fail. If you need to send a messages during startup, implement `ApplicationListener` and wait for the `ContextRefreshedEvent`. Alternatively, implement `SmartLifecycle`, put your bean in a late phase, and send the messages from the `start()` method.

5.6.1. Considerations When Using Packaged (for example, Shaded) Jars

Spring Integration bootstraps certain features by using Spring Framework's `SpringFactories` mechanism to load several `IntegrationConfigurationInitializer` classes. This includes the `-core` jar as well as certain others, including `-http` and `-jmx`. The information for this process is stored in a `META-INF/spring.factories` file in each jar.

Some developers prefer to repackage their application and all dependencies into a single jar by using well known tools, such as the Apache Maven Shade Plugin.

By default, the shade plugin does not merge the `spring.factories` files when producing the shaded jar.

In addition to `spring.factories`, other `META-INF` files (`spring.handlers` and `spring.schemas`) are used for XML configuration. These files also need to be merged.

Spring Boot's executable jar mechanism takes a different approach, in that it nests the jars, thus retaining each `spring.factories` file on the class path. So, with a Spring Boot application, nothing more is needed if you use its default executable jar format.

Even if you do not use Spring Boot, you can still use the tooling provided by Boot to enhance the shade plugin by adding transformers for the above mentioned files.

You may wish to consult the current spring-boot-starter-parent pom to see the current settings that boot uses. The following example shows how to configure the plugin:

Example 1. pom.xml

```
...
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <configuration>
                <keepDependenciesWithProvidedScope>
true</keepDependenciesWithProvidedScope>
                <createDependencyReducedPom>true</createDependencyReducedPom>
            </configuration>
            <dependencies>
                <dependency> ①
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>
                    <version>${spring.boot.version}</version>
                </dependency>
            </dependencies>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <transformers> ②
                            <transformer
                                implementation=
"org.apache.maven.plugins.shade.resource.AppendingTransformer">
                                <resource>META-INF/spring.handlers</resource>
                            </transformer>
                            <transformer
                                implementation=
"org.springframework.boot.maven.PropertiesMergingResourceTransformer">
                                <resource>META-INF/spring.factories</resource>
                            </transformer>
                            <transformer
                                implementation=
"org.apache.maven.plugins.shade.resource.AppendingTransformer">
                                <resource>META-INF/spring.schemas</resource>
                            </transformer>
                            <transformer
                                implementation=
"org.apache.maven.plugins.shade.resource.ServicesResourceTransformer" />
                        </transformers>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
...
```

Specifically,

① Add the `spring-boot-maven-plugin` as a dependency.

② Configure the transformers.

You can add a property for `${spring.boot.version}` or use an explicit version.

5.7. Programming Tips and Tricks

This section documents some of the ways to get the most from Spring Integration.

5.7.1. XML Schemas

When using XML configuration, to avoid getting false schema validation errors, you should use a "Spring-aware" IDE, such as the Spring Tool Suite (STS), Eclipse with the Spring IDE plugins, or IntelliJ IDEA. These IDEs know how to resolve the correct XML schema from the classpath (by using the `META-INF/spring.schemas` file in the jars). When using STS or Eclipse with the plugin, you must enable `Spring Project Nature` on the project.

The schemas hosted on the internet for certain legacy modules (those that existed in version 1.0) are the 1.0 versions for compatibility reasons. If your IDE uses these schemas, you are likely to see false errors.

Each of these online schemas has a warning similar to the following:

This schema is for the 1.0 version of Spring Integration Core. We cannot update it to the current schema because that will break any applications using 1.0.3 or lower. For subsequent versions, the unversioned schema is resolved from the classpath and obtained from the jar. Please refer to github:

github.com/spring-projects/spring-integration/tree/master/spring-integration-core/src/main/resources/org/springframework/integration/config

The affected modules are

- `core` (`spring-integration.xsd`)
- `file`
- `http`
- `jms`
- `mail`
- `rmi`
- `security`
- `stream`
- `ws`

- `xml`

5.7.2. Finding Class Names for Java and DSL Configuration

With XML configuration and Spring Integration Namespace support, the XML parsers hide how target beans are declared and wired together. For Java configuration, it is important to understand the Framework API for target end-user applications.

The first-class citizens for EIP implementation are `Message`, `Channel`, and `Endpoint` (see Main Components, earlier in this chapter). Their implementations (contracts) are:

- `org.springframework.messaging.Message`: See Message;
- `org.springframework.messaging.MessageChannel`: See Message Channels;
- `org.springframework.integration.endpoint.AbstractEndpoint`: See Poller.

The first two are simple enough to understand how to implement, configure, and use. The last one deserves more attention

The `AbstractEndpoint` is widely used throughout the Spring Framework for different component implementations. Its main implementations are:

- `EventDrivenConsumer`, used when we subscribe to a `SubscribableChannel` to listen for messages.
- `PollingConsumer`, used when we poll for messages from a `PollableChannel`.

When you use messaging annotations or the Java DSL, you need to worry about these components, because the Framework automatically produces them with appropriate annotations and `BeanPostProcessor` implementations. When building components manually, you should use the `ConsumerEndpointFactoryBean` to help determine the target `AbstractEndpoint` consumer implementation to create, based on the provided `inputChannel` property.

On the other hand, the `ConsumerEndpointFactoryBean` delegates to an another first class citizen in the Framework - `org.springframework.messaging.MessageHandler`. The goal of the implementation of this interface is to handle the message consumed by the endpoint from the channel. All EIP components in Spring Integration are `MessageHandler` implementations (for example, `AggregatingMessageHandler`, `MessageTransformingHandler`, `AbstractMessageSplitter`, and others). The target protocol outbound adapters (`FileWritingMessageHandler`, `HttpRequestExecutingMessageHandler`, `AbstractMqttMessageHandler`, and others) are also `MessageHandler` implementations. When you develop Spring Integration applications with Java configuration, you should look into the Spring Integration module to find an appropriate `MessageHandler` implementation to use for the `@ServiceActivator` configuration. For example, to send an XMPP message (see XMPP Support) you should configure something like the following:

```
@Bean
@ServiceActivator(inputChannel = "input")
public MessageHandler sendChatMessageHandler(XMPPConnection xmppConnection) {
    ChatMessageSendingMessageHandler handler = new
ChatMessageSendingMessageHandler(xmppConnection);

    DefaultXmppHeaderMapper xmppHeaderMapper = new DefaultXmppHeaderMapper();
    xmppHeaderMapper.setRequestHeaderNames("*");
    handler.setHeaderMapper(xmppHeaderMapper);

    return handler;
}
```

The `MessageHandler` implementations represent the outbound and processing part of the message flow.

The inbound message flow side has its own components, which are divided into polling and listening behaviors. The listening (message-driven) components are simple and typically require only one target class implementation to be ready to produce messages. Listening components can be one-way `MessageProducerSupport` implementations, (such as `AbstractMqttMessageDrivenChannelAdapter` and `ImapIdleChannelAdapter`) or request-reply `MessagingGatewaySupport` implementations (such as `AmqpInboundGateway` and `AbstractWebServiceInboundGateway`).

Polling inbound endpoints are for those protocols that do not provide a listener API or are not intended for such a behavior, including any file based protocol (such as FTP), any data bases (RDBMS or NoSQL), and others.

These inbound endpoints consist of two components: the poller configuration, to initiate the polling task periodically, and a message source class to read data from the target protocol and produce a message for the downstream integration flow. The first class for the poller configuration is a `SourcePollingChannelAdapter`. It is one more `AbstractEndpoint` implementation, but especially for polling to initiate an integration flow. Typically, with the messaging annotations or Java DSL, you should not worry about this class. The Framework produces a bean for it, based on the `@InboundChannelAdapter` configuration or a Java DSL builder spec.

Message source components are more important for the target application development, and they all implement the `MessageSource` interface (for example, `MongoDbMessageSource` and `AbstractTwitterMessageSource`). With that in mind, our config for reading data from an RDBMS table with JDBC could resemble the following:

```
@Bean
@InboundChannelAdapter(value = "fooChannel", poller = @Poller(fixedDelay="5000"))
public MessageSource<?> storedProc(DataSource dataSource) {
    return new JdbcPollingChannelAdapter(dataSource, "SELECT * FROM foo where
status = 0");
}
```

You can find all the required inbound and outbound classes for the target protocols in the particular Spring Integration module (in most cases, in the respective package). For example, the `spring-integration-websocket` adapters are:

- `o.s.i.websocket.inbound.WebSocketInboundChannelAdapter`: Implements `MessageProducerSupport` to listen for frames on the socket and produce message to the channel.
- `o.s.i.websocket.outbound.WebSocketOutboundMessageHandler`: The one-way `AbstractMessageHandler` implementation to convert incoming messages to the appropriate frame and send over websocket.

If you are familiar with Spring Integration XML configuration, starting with version 4.3, we provide information in the XSD element definitions about which target classes are used to declare beans for the adapter or gateway, as the following example shows:

```
<xsd:element name="outbound-async-gateway">
    <xsd:annotation>
        <xsd:documentation>
Configures a Consumer Endpoint for the
'o.s.i.amqp.outbound.AsyncAmqpOutboundGateway'
that will publish an AMQP Message to the provided Exchange and expect a reply
Message.
The sending thread returns immediately; the reply is sent asynchronously; uses
'AsyncRabbitTemplate.sendAndReceive()'.
        </xsd:documentation>
    </xsd:annotation>
```

5.8. POJO Method invocation

As discussed in Programming Considerations, we recommend using a POJO programming style, as the following example shows:

```
@ServiceActivator
public String myService(String payload) { ... }
```

In this case, the framework extracts a `String` payload, invokes your method, and wraps the result in a message to send to the next component in the flow (the original headers are copied to the new message). In fact, if you use XML configuration, you do not even need the `@ServiceActivator` annotation, as the following paired examples show:

```
<int:service-activator ... ref="myPojo" method="myService" />
```

```
public String myService(String payload) { ... }
```

You can omit the `method` attribute as long as there is no ambiguity in the public methods on the class.

You can also obtain header information in your POJO methods, as the following example shows:

```
@ServiceActivator
public String myService(@Payload String payload, @Header("foo") String fooHeader)
{ ... }
```

You can also dereference properties on the message, as the following example shows:

```
@ServiceActivator
public String myService(@Payload("payload.foo") String foo, @Header("bar.baz")
String barbaz) { ... }
```

Because various POJO method invocations are available, versions prior to 5.0 used SpEL (Spring Expression Language) to invoke the POJO methods. SpEL (even interpreted) is usually "fast enough" for these operations, when compared to the actual work usually done in the methods. However, starting with version 5.0, the `org.springframework.messaging.handler.invocation.InvocableHandlerMethod` is used by default whenever possible. This technique is usually faster to execute than interpreted SpEL and is consistent with other Spring messaging projects. The `InvocableHandlerMethod` is similar to the technique used to invoke controller methods in Spring MVC. There are certain methods that are still always invoked when using SpEL. Examples include annotated parameters with dereferenced properties, as discussed earlier. This is because SpEL has the capability to navigate a property path.

There may be some other corner cases that we have not considered that also do not work with `InvocableHandlerMethod` instances. For this reason, we automatically fall back to using SpEL in those cases.

If you wish, you can also set up your POJO method such that it always uses SpEL, with the

`UseSpelInvoker` annotation, as the following example shows:

```
@UseSpelInvoker(compilerMode = "IMMEDIATE")
public void bar(String bar) { ... }
```

If the `compilerMode` property is omitted, the `spring.expression.compiler.mode` system property determines the compiler mode. See SpEL compilation for more information about compiled SpEL.

Core Messaging

This section covers all aspects of the core messaging API in Spring Integration. It covers messages, message channels, and message endpoints. It also covers many of the enterprise integration patterns, such as filter, router, transformer, service activator , splitter, and aggregator.

This section also contains material about system management, including the control bus and message history support.

Chapter 6. Messaging Channels

6.1. Message Channels

While the `Message` plays the crucial role of encapsulating data, it is the `MessageChannel` that decouples message producers from message consumers.

6.1.1. The MessageChannel Interface

Spring Integration's top-level `MessageChannel` interface is defined as follows:

```
public interface MessageChannel {

    boolean send(Message message);

    boolean send(Message message, long timeout);
}
```

When sending a message, the return value is `true` if the message is sent successfully. If the send call times out or is interrupted, it returns `false`.

`PollableChannel`

Since message channels may or may not buffer messages (as discussed in the Spring Integration Overview), two sub-interfaces define the buffering (pollable) and non-buffering (subscribable) channel behavior. The following listing shows the definition of the `PollableChannel` interface:

```
public interface PollableChannel extends MessageChannel {

    Message<?> receive();

    Message<?> receive(long timeout);

}
```

As with the send methods, when receiving a message, the return value is null in the case of a timeout or interrupt.

`SubscribableChannel`

The `SubscribableChannel` base interface is implemented by channels that send messages directly to their subscribed `MessageHandler` instances. Therefore, they do not provide receive methods for polling. Instead, they define methods for managing those subscribers. The following listing shows the definition of the `SubscribableChannel` interface:

```
public interface SubscribableChannel extends MessageChannel {

    boolean subscribe(MessageHandler handler);

    boolean unsubscribe(MessageHandler handler);

}
```

6.1.2. Message Channel Implementations

Spring Integration provides several different message channel implementations. The following sections briefly describe each one.

`PublishSubscribeChannel`

The `PublishSubscribeChannel` implementation broadcasts any `Message` sent to it to all of its subscribed handlers. This is most often used for sending event messages, whose primary role is notification (as opposed to document messages, which are generally intended to be processed by a single handler). Note that the `PublishSubscribeChannel` is intended for sending only. Since it broadcasts to its subscribers directly when its `send(Message)` method is invoked, consumers cannot poll for messages (it does not implement `PollableChannel` and therefore has no `receive()` method). Instead, any subscriber must itself be a `MessageHandler`, and the subscriber's `handleMessage(Message)` method is invoked in turn.

Prior to version 3.0, invoking the `send` method on a `PublishSubscribeChannel` that had no subscribers returned `false`. When used in conjunction with a `MessagingTemplate`, a `MessageDeliveryException` was thrown. Starting with version 3.0, the behavior has changed such that a `send` is always considered successful if at least the minimum subscribers are present (and successfully handle the message). This behavior can be modified by setting the `minSubscribers` property, which defaults to `0`.

If you use a `TaskExecutor`, only the presence of the correct number of subscribers is used for this determination, because the actual handling of the message is performed asynchronously.

`QueueChannel`

The `QueueChannel` implementation wraps a queue. Unlike the `PublishSubscribeChannel`, the `QueueChannel` has point-to-point semantics. In other words, even if the channel has multiple consumers, only one of them should receive any `Message` sent to that channel. It provides a default no-argument constructor (providing an essentially unbounded capacity of `Integer.MAX_VALUE`) as well as a constructor that accepts the queue capacity, as the following listing shows:

```
public QueueChannel(int capacity)
```

A channel that has not reached its capacity limit stores messages in its internal queue, and the `send(Message<?>)` method returns immediately, even if no receiver is ready to handle the message. If the queue has reached capacity, the sender blocks until room is available in the queue. Alternatively, if you use the send method that has an additional timeout parameter, the queue blocks until either room is available or the timeout period elapses, whichever occurs first. Similarly, a `receive()` call returns immediately if a message is available on the queue, but, if the queue is empty, then a receive call may block until either a message is available or the timeout, if provided, elapses. In either case, it is possible to force an immediate return regardless of the queue's state by passing a timeout value of 0. Note, however, that calls to the versions of `send()` and `receive()` with no `timeout` parameter block indefinitely.

`PriorityChannel`

Whereas the `QueueChannel` enforces first-in-first-out (FIFO) ordering, the `PriorityChannel` is an alternative implementation that allows for messages to be ordered within the channel based upon a priority. By default, the priority is determined by the `priority` header within each message. However, for custom priority determination logic, a comparator of type `Comparator<Message<?>>` can be provided to the `PriorityChannel` constructor.

`RendezvousChannel`

The `RendezvousChannel` enables a "direct-handoff" scenario, wherein a sender blocks until another party invokes the channel's `receive()` method. The other party blocks until the sender sends the message. Internally, this implementation is quite similar to the `QueueChannel`, except that it uses a `SynchronousQueue` (a zero-capacity implementation of `BlockingQueue`). This works well in situations where the sender and receiver operate in different threads, but asynchronously dropping the message in a queue is not appropriate. In other words, with a `RendezvousChannel`, the sender knows that some receiver has accepted the message, whereas with a `QueueChannel`, the message would have been stored to the internal queue and potentially never received.

Keep in mind that all of these queue-based channels are storing messages in-memory only by default. When persistence is required, you can either provide a 'message-store' attribute within the 'queue' element to reference a persistent `MessageStore` implementation or you can replace the local channel with one that is backed by a persistent broker, such as a JMS-backed channel or channel adapter. The latter option lets you take advantage of any JMS provider's implementation for message persistence, as discussed in JMS Support. However, when buffering in a queue is not necessary, the simplest approach is to rely upon the `DirectChannel`, discussed in the next section.

The `RendezvousChannel` is also useful for implementing request-reply operations. The sender can create a temporary, anonymous instance of `RendezvousChannel`, which it then sets as the 'replyChannel' header when building a `Message`. After sending that `Message`, the sender can immediately call `receive` (optionally providing a timeout value) in order to block while waiting for a reply `Message`. This is very similar to the implementation used internally by many of Spring Integration's request-reply components.

DirectChannel

The DirectChannel has point-to-point semantics but otherwise is more similar to the PublishSubscribeChannel than any of the queue-based channel implementations described earlier. It implements the SubscribableChannel interface instead of the PollableChannel interface, so it dispatches messages directly to a subscriber. As a point-to-point channel, however, it differs from the PublishSubscribeChannel in that it sends each Message to a single subscribed MessageHandler.

In addition to being the simplest point-to-point channel option, one of its most important features is that it enables a single thread to perform the operations on "both sides" of the channel. For example, if a handler subscribes to a DirectChannel, then sending a Message to that channel triggers invocation of that handler's handleMessage(Message) method directly in the sender's thread, before the send() method invocation can return.

The key motivation for providing a channel implementation with this behavior is to support transactions that must span across the channel while still benefiting from the abstraction and loose coupling that the channel provides. If the send call is invoked within the scope of a transaction, the outcome of the handler's invocation (for example, updating a database record) plays a role in determining the ultimate result of that transaction (commit or rollback).

Since the DirectChannel is the simplest option and does not add any additional overhead that would be required for scheduling and managing the threads of a poller, it is the default channel type within Spring Integration. The general idea is to define the channels for an application, consider which of those need to provide buffering or to throttle input, and modify those to be queue-based PollableChannels. Likewise, if a channel needs to broadcast messages, it should not be a DirectChannel but rather a PublishSubscribeChannel. Later, we show how each of these channels can be configured.

The DirectChannel internally delegates to a message dispatcher to invoke its subscribed message handlers, and that dispatcher can have a load-balancing strategy exposed by load-balancer or load-balancer-ref attributes (mutually exclusive). The load balancing strategy is used by the message dispatcher to help determine how messages are distributed amongst message handlers when multiple message handlers subscribe to the same channel. As a convenience, the load-balancer attribute exposes an enumeration of values pointing to pre-existing implementations of LoadBalancingStrategy. round-robin (load-balances across the handlers in rotation) and none (for the cases where one wants to explicitly disable load balancing) are the only available values. Other strategy implementations may be added in future versions. However, since version 3.0, you can provide your own implementation of the LoadBalancingStrategy and inject it by using the load-balancer-ref attribute, which should point to a bean that implements LoadBalancingStrategy, as the following example shows:

A FixedSubscriberChannel is a SubscribableChannel that only supports a single MessageHandler subscriber that cannot be unsubscribed. This is useful for high-throughput performance use-cases when no other subscribers are involved and no channel interceptors are needed.

```
<int:channel id="lbRefChannel">
  <int:dispatcher load-balancer-ref="lb"/>
</int:channel>

<bean id="lb" class="foo.bar.SampleLoadBalancingStrategy"/>
```

Note that the `load-balancer` and `load-balancer-ref` attributes are mutually exclusive.

The load-balancing also works in conjunction with a boolean `failover` property. If the `failover` value is true (the default), the dispatcher falls back to any subsequent handlers (as necessary) when preceding handlers throw exceptions. The order is determined by an optional order value defined on the handlers themselves or, if no such value exists, the order in which the handlers subscribed.

If a certain situation requires that the dispatcher always try to invoke the first handler and then fall back in the same fixed order sequence every time an error occurs, no load-balancing strategy should be provided. In other words, the dispatcher still supports the `failover` boolean property even when no load-balancing is enabled. Without load-balancing, however, the invocation of handlers always begins with the first, according to their order. For example, this approach works well when there is a clear definition of primary, secondary, tertiary, and so on. When using the namespace support, the `order` attribute on any endpoint determines the order.

Keep in mind that load-balancing and `failover` apply only when a channel has more than one subscribed message handler. When using the namespace support, this means that more than one endpoint shares the same channel reference defined in the `input-channel` attribute.

Starting with version 5.2, when `failover` is true, a failure of the current handler together with the failed message is logged under `debug` or `info` if configured respectively.

`ExecutorChannel`

The `ExecutorChannel` is a point-to-point channel that supports the same dispatcher configuration as `DirectChannel` (load-balancing strategy and the `failover` boolean property). The key difference between these two dispatching channel types is that the `ExecutorChannel` delegates to an instance of `TaskExecutor` to perform the dispatch. This means that the send method typically does not block, but it also means that the handler invocation may not occur in the sender's thread. It therefore does not support transactions that span the sender and receiving handler.

The sender can sometimes block. For example, when using a `TaskExecutor` with a rejection policy that throttles the client (such as the `ThreadPoolExecutor.CallerRunsPolicy`), the sender's thread can execute the method any time the thread pool is at its maximum capacity and the executor's work queue is full. Since that situation would only occur in a non-predictable way, you should not rely upon it for transactions.

`FluxMessageChannel`

The `FluxMessageChannel` is an `org.reactivestreams.Publisher` implementation for `"sinking"` sent messages into an internal `reactor.core.publisher.Flux` for on demand consumption by reactive subscribers downstream. This channel implementation is neither a `SubscribableChannel`, nor a `PollableChannel`, so only `org.reactivestreams.Subscriber` instances can be used to consume from this channel honoring back-pressure nature of reactive streams. On the other hand, the `FluxMessageChannel` implements a `ReactiveStreamsSubscribableChannel` with its `subscribeTo(Publisher<Message<?>>)` contract allowing receiving events from reactive source publishers, bridging a reactive stream into the integration flow. To achieve fully reactive behavior for the whole integration flow, such a channel must be placed between all the endpoints in the flow.

See Reactive Streams Support for more information about interaction with Reactive Streams.

Scoped Channel

Spring Integration 1.0 provided a `ThreadLocalChannel` implementation, but that has been removed as of 2.0. Now the more general way to handle the same requirement is to add a `scope` attribute to a channel. The value of the attribute can be the name of a scope that is available within the context. For example, in a web environment, certain scopes are available, and any custom scope implementations can be registered with the context. The following example shows a thread-local scope being applied to a channel, including the registration of the scope itself:

```
<int:channel id="threadScopedChannel" scope="thread">
     <int:queue />
</int:channel>

<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
    <property name="scopes">
        <map>
            <entry key="thread" value=
"org.springframework.context.support.SimpleThreadScope" />
        </map>
    </property>
</bean>
```

The channel defined in the previous example also delegates to a queue internally, but the channel is bound to the current thread, so the contents of the queue are similarly bound. That way, the thread that sends to the channel can later receive those same messages, but no other thread would be able to access them. While thread-scoped channels are rarely needed, they can be useful in situations where `DirectChannel` instances are being used to enforce a single thread of operation but any reply messages should be sent to a “terminal” channel. If that terminal channel is thread-scoped, the original sending thread can collect its replies from the terminal channel.

Now, since any channel can be scoped, you can define your own scopes in addition to thread-Local.

6.1.3. Channel Interceptors

One of the advantages of a messaging architecture is the ability to provide common behavior and capture meaningful information about the messages passing through the system in a non-invasive way. Since the `Message` instances are sent to and received from `MessageChannel` instances, those channels provide an opportunity for intercepting the send and receive operations. The `ChannelInterceptor` strategy interface, shown in the following listing, provides methods for each of those operations:

```
public interface ChannelInterceptor {

    Message<?> preSend(Message<?> message, MessageChannel channel);

    void postSend(Message<?> message, MessageChannel channel, boolean sent);

    void afterSendCompletion(Message<?> message, MessageChannel channel, boolean
sent, Exception ex);

    boolean preReceive(MessageChannel channel);

    Message<?> postReceive(Message<?> message, MessageChannel channel);

    void afterReceiveCompletion(Message<?> message, MessageChannel channel,
Exception ex);
}
```

After implementing the interface, registering the interceptor with a channel is just a matter of making the following call:

```
channel.addInterceptor(someChannelInterceptor);
```

The methods that return a `Message` instance can be used for transforming the `Message` or can return 'null' to prevent further processing (of course, any of the methods can throw a `RuntimeException`). Also, the `preReceive` method can return `false` to prevent the receive operation from proceeding.

Keep in mind that `receive()` calls are only relevant for `PollableChannels`. In fact, the `SubscribableChannel` interface does not even define a `receive()` method. The reason for this is that when a `Message` is sent to a `SubscribableChannel`, it is sent directly to zero or more subscribers, depending on the type of channel (for example, a `PublishSubscribeChannel` sends to all of its subscribers). Therefore, the `preReceive(…)`, `postReceive(…)`, and `afterReceiveCompletion(…)` interceptor methods are invoked only when the interceptor is applied to a `PollableChannel`.

Spring Integration also provides an implementation of the Wire Tap pattern. It is a simple

interceptor that sends the `Message` to another channel without otherwise altering the existing flow. It can be very useful for debugging and monitoring. An example is shown in Wire Tap.

Because it is rarely necessary to implement all of the interceptor methods, the interface provides no-op methods (methods returning `void` method have no code, the `Message`-returning methods return the `Message` as-is, and the `boolean` method returns `true`).

The order of invocation for the interceptor methods depends on the type of channel. As described earlier, the queue-based channels are the only ones where the receive method is intercepted in the first place. Additionally, the relationship between send and receive interception depends on the timing of the separate sender and receiver threads. For example, if a receiver is already blocked while waiting for a message, the order could be as follows: `preSend`, `preReceive`, `postReceive`, `postSend`. However, if a receiver polls after the sender has placed a message on the channel and has already returned, the order would be as follows: `preSend`, `postSend` (some-time-elapses), `preReceive`, `postReceive`. The time that elapses in such a case depends on a number of factors and is therefore generally unpredictable (in fact, the receive may never happen). The type of queue also plays a role (for example, rendezvous versus priority). In short, you cannot rely on the order beyond the fact that `preSend` precedes `postSend` and `preReceive` precedes `postReceive`.

Starting with Spring Framework 4.1 and Spring Integration 4.1, the `ChannelInterceptor` provides new methods: `afterSendCompletion()` and `afterReceiveCompletion()`. They are invoked after `send()' and 'receive()` calls, regardless of any exception that is raised, which allow for resource cleanup. Note that the channel invokes these methods on the `ChannelInterceptor` list in the reverse order of the initial `preSend()` and `preReceive()` calls.

Starting with version 5.1, global channel interceptors now apply to dynamically registered channels - such as through beans that are initialized by using `beanFactory.initializeBean()` or `IntegrationFlowContext` when using the Java DSL. Previously, interceptors were not applied when beans were created after the application context was refreshed.

Also, starting with version 5.1, `ChannelInterceptor.postReceive()` is no longer called when no message is received; it is no longer necessary to check for a `null Message<?>`. Previously, the method was called. If you have an interceptor that relies on the previous behavior, implement `afterReceiveCompleted()` instead, since that method is invoked, regardless of whether a message is received or not.

Starting with version 5.2, the `ChannelInterceptorAware` is deprecated in favor of `InterceptableChannel` from the Spring Messaging module, which it extends now for backward compatibility.

6.1.4. `MessagingTemplate`

When the endpoints and their various configuration options are introduced, Spring Integration provides a foundation for messaging components that enables non-invasive invocation of your application code from the messaging system. However, it is sometimes necessary to invoke the

messaging system from your application code. For convenience when implementing such use cases, Spring Integration provides a `MessagingTemplate` that supports a variety of operations across the message channels, including request and reply scenarios. For example, it is possible to send a request and wait for a reply, as follows:

```
MessagingTemplate template = new MessagingTemplate();

Message reply = template.sendAndReceive(someChannel, new GenericMessage("test"));
```

In the preceding example, a temporary anonymous channel would be created internally by the template. The 'sendTimeout' and 'receiveTimeout' properties may also be set on the template, and other exchange types are also supported. The following listing shows the signatures for such methods:

```
public boolean send(final MessageChannel channel, final Message<?> message) { ...
}

public Message<?> sendAndReceive(final MessageChannel channel, final Message<?>
request) { ...
}

public Message<?> receive(final PollableChannel<?> channel) { ...
}
```

A less invasive approach that lets you invoke simple interfaces with payload or header values instead of `Message` instances is described in Enter the `GatewayProxyFactoryBean`.

6.1.5. Configuring Message Channels

To create a message channel instance, you can use the `<channel/>` element, as follows:

```
<int:channel id="exampleChannel"/>
```

The equivalent Java configuration declares a `DirectChannel` `@Bean`:

```
@Bean
public MessageChannel exampleChannel() {
    return new DirectChannel();
}
```

The default channel type is point-to-point. To create a publish-subscribe channel, use the `<publish-subscribe-channel/>` element, as follows:

```
<int:publish-subscribe-channel id="exampleChannel"/>
```

The Java configuration is:

```
@Bean
public MessageChannel exampleChannel() {
    return new PublishSubscribeChannel();
}
```

When you use the `<channel/>` element without any sub-elements, it creates a `DirectChannel` instance (a `SubscribableChannel`).

However, you can alternatively provide a variety of `<queue/>` sub-elements to create any of the pollable channel types (as described in Message Channel Implementations). The following sections shows examples of each channel type.

`DirectChannel` Configuration

As mentioned earlier, `DirectChannel` is the default type. The following listing shows who to define one:

```
<int:channel id="directChannel"/>
```

In Java Configuration:

```
@Bean
public MessageChannel directChannel() {
    return new DirectChannel();
}
```

A default channel has a round-robin load-balancer and also has failover enabled (see `DirectChannel` for more detail). To disable one or both of these, add a `<dispatcher/>` sub-element and configure the attributes as follows:

```
<int:channel id="failFastChannel">
    <int:dispatcher failover="false"/>
</channel>

<int:channel id="channelWithFixedOrderSequenceFailover">
    <int:dispatcher load-balancer="none"/>
</int:channel>
```

In Java Configuration:

```
@Bean
public MessageChannel failFastChannel() {
    DirectChannel channel = new DirectChannel();
    channel.setFailover(false);
    return channel;
}

@Bean
public MessageChannel failFastChannel() {
    return new DirectChannel(null);
}
```

Datatype Channel Configuration

Sometimes, a consumer can process only a particular type of payload, forcing you to ensure the payload type of the input messages. The first thing that comes to mind may be to use a message filter. However, all that message filter can do is filter out messages that are not compliant with the requirements of the consumer. Another way would be to use a content-based router and route messages with non-compliant data-types to specific transformers to enforce transformation and conversion to the required data type. This would work, but a simpler way to accomplish the same thing is to apply the Datatype Channel pattern. You can use separate datatype channels for each specific payload data type.

To create a datatype channel that accepts only messages that contain a certain payload type, provide the data type's fully-qualified class name in the channel element's `datatype` attribute, as the following example shows:

```
<int:channel id="numberChannel" datatype="java.lang.Number"/>
```

In Java Configuration:

```
@Bean
public MessageChannel numberChannel() {
    DirectChannel channel = new DirectChannel();
    channel.setDatatypes(Number.class);
    return channel;
}
```

Note that the type check passes for any type that is assignable to the channel's datatype. In other words, the `numberChannel` in the preceding example would accept messages whose payload is `java.lang.Integer` or `java.lang.Double`. Multiple types can be provided as a comma-delimited list, as the following example shows:

```
<int:channel id="stringOrNumberChannel" datatype=
"java.lang.String,java.lang.Number"/>
```

So the 'numberChannel' in the preceding example accepts only messages with a data type of `java.lang.Number`. But what happens if the payload of the message is not of the required type? It depends on whether you have defined a bean named `integrationConversionService` that is an instance of Spring's Conversion Service. If not, then an `Exception` would be thrown immediately. However, if you have defined an `integrationConversionService` bean, it is used in an attempt to convert the message's payload to the acceptable type.

You can even register custom converters. For example, suppose you send a message with a `String` payload to the 'numberChannel' we configured above. You might handle the message as follows:

```
MessageChannel inChannel = context.getBean("numberChannel", MessageChannel.class);
inChannel.send(new GenericMessage<String>("5"));
```

Typically this would be a perfectly legal operation. However, since we use Datatype Channel, the result of such operation would generate an exception similar to the following:

```
Exception in thread "main"
org.springframework.integration.MessageDeliveryException:
Channel 'numberChannel'
expected one of the following datataypes [class java.lang.Number],
but received [class java.lang.String]
...
```

The exception happens because we require the payload type to be a `Number`, but we sent a `String`. So we need something to convert a `String` to a `Number`. For that, we can implement a converter similar to the following example:

```
public static class StringToIntegerConverter implements Converter<String, Integer>
{
    public Integer convert(String source) {
        return Integer.parseInt(source);
    }
}
```

Then we can register it as a converter with the Integration Conversion Service, as the following example shows:

```
<int:converter ref="strToInt"/>

<bean id="strToInt" class=
"org.springframework.integration.util.Demo.StringToIntegerConverter"/>
```

With Java Configuration you must use an `@IntegrationConverter` next to a `@Bean` annotation:

```
@Bean
@IntegrationConverter
public StringToIntegerConverter strToInt {
    return new StringToIntegerConverter();
}
```

Or on the `StringToIntegerConverter` class when it is marked with the `@Component` annotation for auto-scanning.

When the 'converter' element is parsed, it creates the `integrationConversionService` bean if one is not already defined. With that converter in place, the `send` operation would now be successful,

because the datatype channel uses that converter to convert the `String` payload to an `Integer`.

For more information regarding payload type conversion, see Payload Type Conversion.

Beginning with version 4.0, the `integrationConversionService` is invoked by the `DefaultDatatypeChannelMessageConverter`, which looks up the conversion service in the application context. To use a different conversion technique, you can specify the `message-converter` attribute on the channel. This must be a reference to a `MessageConverter` implementation. Only the `fromMessage` method is used. It provides the converter with access to the message headers (in case the conversion might need information from the headers, such as `content-type`). The method can return only the converted payload or a full `Message` object. If the latter, the converter must be careful to copy all the headers from the inbound message.

Alternatively, you can declare a `<bean/>` of type `MessageConverter` with an ID of `datatypeChannelMessageConverter`, and that converter is used by all channels with a `datatype`.

`QueueChannel` Configuration

To create a `QueueChannel`, use the `<queue/>` sub-element. You may specify the channel's capacity as follows:

```
<int:channel id="queueChannel">
    <queue capacity="25"/>
</int:channel>
```

If you do not provide a value for the 'capacity' attribute on this `<queue/>` sub-element, the resulting queue is unbounded. To avoid issues such as running out of memory, we highly recommend that you set an explicit value for a bounded queue.

With Java Configuration:

```
@Bean
public PollableChannel queueChannel() {
    return new QueueChannel(25);
}
```

Persistent `QueueChannel` Configuration

Since a `QueueChannel` provides the capability to buffer messages but does so in-memory only by default, it also introduces a possibility that messages could be lost in the event of a system failure. To mitigate this risk, a `QueueChannel` may be backed by a persistent implementation of the `MessageGroupStore` strategy interface. For more details on `MessageGroupStore` and `MessageStore`, see Message Store.

The `capacity` attribute is not allowed when the `message-store` attribute is used.

When a `QueueChannel` receives a `Message`, it adds the message to the message store. When a `Message` is polled from a `QueueChannel`, it is removed from the message store.

By default, a `QueueChannel` stores its messages in an in-memory queue, which can lead to the lost message scenario mentioned earlier. However, Spring Integration provides persistent stores, such as the `JdbcChannelMessageStore`.

You can configure a message store for any `QueueChannel` by adding the `message-store` attribute, as the following example shows:

```
<int:channel id="dbBackedChannel">
    <int:queue message-store="channelStore"/>
</int:channel>

<bean id="channelStore" class="o.s.i.jdbc.store.JdbcChannelMessageStore">
    <property name="dataSource" ref="dataSource"/>
    <property name="channelMessageStoreQueryProvider" ref="queryProvider"/>
</bean>
```

(See samples below for Java Configuration options.)

The Spring Integration JDBC module also provides a schema Data Definition Language (DDL) for a number of popular databases. These schemas are located in the org.springframework.integration.jdbc.store.channel package of that module (`spring-integration-jdbc`).

One important feature is that, with any transactional persistent store (such as `JdbcChannelMessageStore`), as long as the poller has a transaction configured, a message removed from the store can be permanently removed only if the transaction completes successfully. Otherwise the transaction rolls back, and the `Message` is not lost.

Many other implementations of the message store are available as the growing number of Spring projects related to “NoSQL” data stores come to provide underlying support for these stores. You can also provide your own implementation of the `MessageGroupStore` interface if you cannot find one that meets your particular needs.

Since version 4.0, we recommend that `QueueChannel` instances be configured to use a `ChannelMessageStore`, if possible. These are generally optimized for this use, as compared to a general message store. If the `ChannelMessageStore` is a `ChannelPriorityMessageStore`, the messages are received in FIFO within priority order. The notion of priority is determined by the message store implementation. For example, the following example shows the Java configuration for the MongoDB Channel Message Store:

```
@Bean
public BasicMessageGroupStore mongoDbChannelMessageStore(MongoDbFactory
mongoDbFactory) {
    MongoDbChannelMessageStore store = new MongoDbChannelMessageStore
(mongoDbFactory);
    store.setPriorityEnabled(true);
    return store;
}

@Bean
public PollableChannel priorityQueue(BasicMessageGroupStore
mongoDbChannelMessageStore) {
    return new PriorityChannel(new MessageGroupQueue(mongoDbChannelMessageStore,
"priorityQueue"));
}
```

Pay attention to the `MessageGroupQueue` class. That is a `BlockingQueue` implementation to use the `MessageGroupStore` operations.

The same implementation with Java DSL might look like the following example:

```
@Bean
public IntegrationFlow priorityFlow(PriorityCapableChannelMessageStore
mongoDbChannelMessageStore) {
    return IntegrationFlows.from((Channels c) ->
            c.priority("priorityChannel", mongoDbChannelMessageStore, "priorityGroup"
))
            ....
            .get();
}
```

Another option to customize the `QueueChannel` environment is provided by the `ref` attribute of the `<int:queue>` sub-element or its particular constructor. This attribute supplies the reference to any `java.util.Queue` implementation. For example, a Hazelcast distributed `IQueue` can be configured as follows:

```
@Bean
public HazelcastInstance hazelcastInstance() {
    return Hazelcast.newHazelcastInstance(new Config()
                                            .setProperty("hazelcast.logging.type",
"log4j"));
}

@Bean
public PollableChannel distributedQueue() {
    return new QueueChannel(hazelcastInstance()
                                .getQueue("springIntegrationQueue"));
}
```

`PublishSubscribeChannel` Configuration

To create a `PublishSubscribeChannel`, use the <publish-subscribe-channel/> element. When using this element, you can also specify the `task-executor` used for publishing messages (if none is specified, it publishes in the sender's thread), as follows:

```
<int:publish-subscribe-channel id="pubsubChannel" task-executor="someExecutor"/>
```

With Java Configuration:

```
@Bean
public MessageChannel pubsubChannel() {
    return new PublishSubscribeChannel(someExecutor());
}
```

If you provide a resequencer or aggregator downstream from a `PublishSubscribeChannel`, you can set the 'apply-sequence' property on the channel to `true`. Doing so indicates that the channel should set the `sequence-size` and `sequence-number` message headers as well as the correlation ID prior to passing along the messages. For example, if there are five subscribers, the `sequence-size` would be set to `5`, and the messages would have `sequence-number` header values ranging from `1` to `5`.

Along with the `Executor`, you can also configure an `ErrorHandler`. By default, the `PublishSubscribeChannel` uses a `MessagePublishingErrorHandler` implementation to send an error to the `MessageChannel` from the `errorChannel` header or into the global `errorChannel` instance. If an `Executor` is not configured, the `ErrorHandler` is ignored and exceptions are thrown directly to the caller's thread.

If you provide a `Resequencer` or `Aggregator` downstream from a `PublishSubscribeChannel`, you can set the 'apply-sequence' property on the channel to `true`. Doing so indicates that the channel should set

the sequence-size and sequence-number message headers as well as the correlation ID prior to passing along the messages. For example, if there are five subscribers, the sequence-size would be set to 5, and the messages would have sequence-number header values ranging from 1 to 5.

The following example shows how to set the apply-sequence header to true:

```
<int:publish-subscribe-channel id="pubsubChannel" apply-sequence="true"/>
```

```
@Bean
public MessageChannel pubsubChannel() {
    PublishSubscribeChannel channel = new PublishSubscribeChannel();
    channel.setApplySequence(false);
    return channel;
}
```

The apply-sequence value is false by default so that a publish-subscribe channel can send the exact same message instances to multiple outbound channels. Since Spring Integration enforces immutability of the payload and header references, when the flag is set to true, the channel creates new Message instances with the same payload reference but different header values.

ExecutorChannel

To create an ExecutorChannel, add the <dispatcher> sub-element with a task-executor attribute. The attribute's value can reference any TaskExecutor within the context. For example, doing so enables configuration of a thread pool for dispatching messages to subscribed handlers. As mentioned earlier, doing so breaks the single-threaded execution context between sender and receiver so that any active transaction context is not shared by the invocation of the handler (that is, the handler may throw an Exception, but the send invocation has already returned successfully). The following example shows how to use the dispatcher element and specify an executor in the task-executor attribute:

```
<int:channel id="executorChannel">
    <int:dispatcher task-executor="someExecutor"/>
</int:channel>
```

In Java Configuration you must use an ExecutorChannel bean definition:

```
@Bean
public MessageChannel executorChannel() {
    return new ExecutorChannel(someExecutor());
}
```

The `load-balancer` and `failover` options are also both available on the <dispatcher/> sub-element, as described earlier in `DirectChannel` Configuration. The same defaults apply. Consequently, the channel has a round-robin load-balancing strategy with failover enabled unless explicit configuration is provided for one or both of those attributes, as the following example shows:

```
<int:channel id="executorChannelWithoutFailover">
    <int:dispatcher task-executor="someExecutor" failover="false"/>
</int:channel>
```

`PriorityChannel` Configuration

To create a `PriorityChannel`, use the `<priority-queue/>` sub-element, as the following example shows:

```
<int:channel id="priorityChannel">
    <int:priority-queue capacity="20"/>
</int:channel>
```

In JavaConfiguration:

```
@Bean
public PollableChannel priorityChannel() {
    return new PriorityChannel(20);
}
```

By default, the channel consults the `priority` header of the message. However, you can instead provide a custom `Comparator` reference. Also, note that the `PriorityChannel` (like the other types) does support the `datatype` attribute. As with the `QueueChannel`, it also supports a `capacity` attribute. The following example demonstrates all of these:

```
<int:channel id="priorityChannel" datatype="example.Widget">
    <int:priority-queue comparator="widgetComparator"
                        capacity="10"/>
</int:channel>
```

```
@Bean
public PollableChannel priorityChannel() {
    PriorityChannel channel = new PriorityChannel(20, widgetComparator());
    channel.setDatatypes(example.Widget.class);
    return channel;
}
```

Since version 4.0, the `priority-channel` child element supports the `message-store` option (`comparator` and `capacity` are not allowed in that case). The message store must be a `PriorityCapableChannelMessageStore`. Implementations of the `PriorityCapableChannelMessageStore` are currently provided for `Redis`, `JDBC`, and `MongoDB`. See `QueueChannel` Configuration and Message Store for more information. You can find sample configuration in Backing Message Channels.

`RendezvousChannel` Configuration

A `RendezvousChannel` is created when the queue sub-element is a `<rendezvous-queue>`. It does not provide any additional configuration options to those described earlier, and its queue does not accept any capacity value, since it is a zero-capacity direct handoff queue. The following example shows how to declare a `RendezvousChannel`:

```
<int:channel id="rendezvousChannel"/>
    <int:rendezvous-queue/>
</int:channel>
```

```
@Bean
public PollableChannel rendezvousChannel() {
    return new RendezvousChannel();
}
```

Scoped Channel Configuration

Any channel can be configured with a `scope` attribute, as the following example shows:

```
<int:channel id="threadLocalChannel" scope="thread"/>
```

Channel Interceptor Configuration

Message channels may also have interceptors, as described in Channel Interceptors. The `<interceptors/>` sub-element can be added to a `<channel/>` (or the more specific element types). You can provide the `ref` attribute to reference any Spring-managed object that implements the `ChannelInterceptor` interface, as the following example shows:

```
<int:channel id="exampleChannel">
    <int:interceptors>
        <ref bean="trafficMonitoringInterceptor"/>
    </int:interceptors>
</int:channel>
```

In general, we recommend defining the interceptor implementations in a separate location, since they usually provide common behavior that can be reused across multiple channels.

Global Channel Interceptor Configuration

Channel interceptors provide a clean and concise way of applying cross-cutting behavior per individual channel. If the same behavior should be applied on multiple channels, configuring the same set of interceptors for each channel would not be the most efficient way. To avoid repeated configuration while also enabling interceptors to apply to multiple channels, Spring Integration provides global interceptors. Consider the following pair of examples:

```
<int:channel-interceptor pattern="input*, thing2*, thing1, !cat*" order="3">
    <bean class="thing1.thing2SampleInterceptor"/>
</int:channel-interceptor>
```

```
<int:channel-interceptor ref="myInterceptor" pattern="input*, thing2*, thing1,
!cat*" order="3"/>

<bean id="myInterceptor" class="thing1.thing2SampleInterceptor"/>
```

Each `<channel-interceptor/>` element lets you define a global interceptor, which is applied on all channels that match any patterns defined by the `pattern` attribute. In the preceding case, the global interceptor is applied on the 'thing1' channel and all other channels that begin with 'thing2' or 'input' but not to channels starting with 'thing3' (since version 5.0).

The addition of this syntax to the pattern causes one possible (though perhaps unlikely) problem. If you have a bean named `!thing1` and you included a pattern of `!thing1` in your channel interceptor's `pattern` patterns, it no longer matches. The pattern now matches all beans not named `thing1`. In this case, you can escape the `!` in the pattern with `\`. The pattern `\!thing1` matches a bean named `!thing1`.

The order attribute lets you manage where this interceptor is injected when there are multiple interceptors on a given channel. For example, channel 'inputChannel' could have individual interceptors configured locally (see below), as the following example shows:

```
<int:channel id="inputChannel">
  <int:interceptors>
    <int:wire-tap channel="logger"/>
  </int:interceptors>
</int:channel>
```

A reasonable question is "how is a global interceptor injected in relation to other interceptors configured locally or through other global interceptor definitions?" The current implementation provides a simple mechanism for defining the order of interceptor execution. A positive number in the `order` attribute ensures interceptor injection after any existing interceptors, while a negative number ensures that the interceptor is injected before existing interceptors. This means that, in the preceding example, the global interceptor is injected after (since its `order` is greater than `0`) the 'wire-tap' interceptor configured locally. If there were another global interceptor with a matching `pattern`, its order would be determined by comparing the values of both interceptors' `order` attributes. To inject a global interceptor before the existing interceptors, use a negative value for the `order` attribute.

Note that both the `order` and `pattern` attributes are optional. The default value for `order` will be 0 and for `pattern`, the default is '*' (to match all channels).

Starting with version 4.3.15, you can configure the `spring.integration.postProcessDynamicBeans = true` property to apply any global interceptors to dynamically created `MessageChannel` beans. See Global Properties for more information.

Wire Tap

As mentioned earlier, Spring Integration provides a simple wire tap interceptor. You can configure a wire tap on any channel within an `<interceptors/>` element. Doing so is especially useful for debugging and can be used in conjunction with Spring Integration's logging channel adapter as follows:

```
<int:channel id="in">
    <int:interceptors>
        <int:wire-tap channel="logger"/>
    </int:interceptors>
</int:channel>

<int:logging-channel-adapter id="logger" level="DEBUG"/>
```

The 'logging-channel-adapter' also accepts an 'expression' attribute so that you can evaluate a SpEL expression against the 'payload' and 'headers' variables. Alternatively, to log the full message `toString()` result, provide a value of `true` for the 'log-full-message' attribute. By default, it is `false` so that only the payload is logged. Setting it to `true` enables logging of all headers in addition to the payload. The 'expression' option provides the most flexibility (for example, `expression="payload.user.name"`).

One of the common misconceptions about the wire tap and other similar components (Message Publishing Configuration) is that they are automatically asynchronous in nature. By default, wire tap as a component is not invoked asynchronously. Instead, Spring Integration focuses on a single unified approach to configuring asynchronous behavior: the message channel. What makes certain parts of the message flow synchronous or asynchronous is the type of Message Channel that has been configured within that flow. That is one of the primary benefits of the message channel abstraction. From the inception of the framework, we have always emphasized the need and the value of the message channel as a first-class citizen of the framework. It is not just an internal, implicit realization of the EIP pattern. It is fully exposed as a configurable component to the end user. So, the wire tap component is only responsible for performing the following tasks:

- Intercept a message flow by tapping into a channel (for example, `channelA`)
- Grab each message
- Send the message to another channel (for example, `channelB`)

It is essentially a variation of the bridge pattern, but it is encapsulated within a channel definition (and hence easier to enable and disable without disrupting a flow). Also, unlike the bridge, it basically forks another message flow. Is that flow synchronous or asynchronous? The answer depends on the type of message channel that 'channelB' is. We have the following options: direct channel, pollable channel, and executor channel. The last two break the thread boundary, making communication over such channels asynchronous, because the dispatching of the message from that channel to its subscribed handlers happens on a different thread than the one used to send the message to that channel. That is what is going to make your wire-tap flow synchronous or asynchronous. It is consistent with other components within the framework (such as message publisher) and adds a level of consistency and simplicity by sparing you from worrying in advance (other than writing thread-safe code) about whether a particular piece of code should be implemented as synchronous or asynchronous. The actual wiring of two pieces of code (say, component A and component B) over a message channel is what makes their collaboration synchronous or asynchronous. You may even want to change from synchronous to asynchronous in

the future, and message channel lets you to do it swiftly without ever touching the code.

One final point regarding the wire tap is that, despite the rationale provided above for not being asynchronous by default, you should keep in mind that it is usually desirable to hand off the message as soon as possible. Therefore, it would be quite common to use an asynchronous channel option as the wire tap's outbound channel. However we doe not enforce asynchronous behavior by default. There are a number of use cases that would break if we did, including that you might not want to break a transactional boundary. Perhaps you use the wire tap pattern for auditing purposes, and you do want the audit messages to be sent within the original transaction. As an example, you might connect the wire tap to a JMS outbound channel adapter. That way, you get the best of both worlds: 1) the sending of a JMS Message can occur within the transaction while 2) it is still a "fire-and-forget" action, thereby preventing any noticeable delay in the main message flow.

Starting with version 4.0, it is important to avoid circular references when an interceptor (such as the `WireTap` class) references a channel. You need to exclude such channels from those being intercepted by the current interceptor. This can be done with appropriate patterns or programmatically. If you have a custom `ChannelInterceptor` that references a `channel`, consider implementing `VetoCapableInterceptor`. That way, the framework asks the interceptor if it is OK to intercept each channel that is a candidate, based on the supplied pattern. You can also add runtime protection in the interceptor methods to ensure that the channel is not one that is referenced by the interceptor. The `WireTap` uses both of these techniques.

Starting with version 4.3, the `WireTap` has additional constructors that take a `channelName` instead of a `MessageChannel` instance. This can be convenient for Java configuration and when channel auto-creation logic is being used. The target `MessageChannel` bean is resolved from the provided `channelName` later, on the first interaction with the interceptor.

Channel resolution requires a `BeanFactory`, so the wire tap instance must be a Spring-managed bean.

This late-binding approach also allows simplification of typical wire-tapping patterns with Java DSL configuration, as the following example shows:

```
@Bean
public PollableChannel myChannel() {
    return MessageChannels.queue()
            .wireTap("loggingFlow.input")
            .get();
}

@Bean
public IntegrationFlow loggingFlow() {
    return f -> f.log();
}
```

Conditional Wire Taps

Wire taps can be made conditional by using the `selector` or `selector-expression` attributes. The `selector` references a `MessageSelector` bean, which can determine at runtime whether the message should go to the tap channel. Similarly, the `selector-expression` is a boolean SpEL expression that performs the same purpose: If the expression evaluates to `true`, the message is sent to the tap channel.

Global Wire Tap Configuration

It is possible to configure a global wire tap as a special case of the Global Channel Interceptor Configuration. To do so, configure a top level `wire-tap` element. Now, in addition to the normal `wire-tap` namespace support, the `pattern` and `order` attributes are supported and work in exactly the same way as they do for the `channel-interceptor`. The following example shows how to configure a global wire tap:

```
<int:wire-tap pattern="input*, thing2*, thing1" order="3" channel="wiretapChannel
"/>
```

A global wire tap provides a convenient way to configure a single-channel wire tap externally without modifying the existing channel configuration. To do so, set the `pattern` attribute to the target channel name. For example, you can use this technique to configure a test case to verify messages on a channel.

6.1.6. Special Channels

If namespace support is enabled, two special channels are defined within the application context by default: `errorChannel` and `nullChannel`. The 'nullChannel' acts like `/dev/null`, logging any message sent to it at the `DEBUG` level and returning immediately. Any time you face channel resolution errors for a reply that you do not care about, you can set the affected component's `output-channel` attribute to 'nullChannel' (the name, 'nullChannel', is reserved within the application context). The 'errorChannel' is used internally for sending error messages and may be overridden with a custom configuration. This is discussed in greater detail in Error Handling.

See also Message Channels in the Java DSL chapter for more information about message channel and interceptors.

6.2. Poller

This section describes how polling works in Spring Integration.

6.2.1. Polling Consumer

When Message Endpoints (Channel Adapters) are connected to channels and instantiated, they produce one of the following instances:

- `PollingConsumer`
- `EventDrivenConsumer`

The actual implementation depends on the type of channel to which these endpoints connect. A channel adapter connected to a channel that implements the `org.springframework.messaging.SubscribableChannel` interface produces an instance of `EventDrivenConsumer`. On the other hand, a channel adapter connected to a channel that implements the `org.springframework.messaging.PollableChannel` interface (such as a `QueueChannel`) produces an instance of `PollingConsumer`.

Polling consumers let Spring Integration components actively poll for Messages rather than process messages in an event-driven manner.

They represent a critical cross-cutting concern in many messaging scenarios. In Spring Integration, polling consumers are based on the pattern with the same name, which is described in the book *Enterprise Integration Patterns*, by Gregor Hohpe and Bobby Woolf. You can find a description of the pattern on the book's website.

6.2.2. Pollable Message Source

Spring Integration offers a second variation of the polling consumer pattern. When inbound channel adapters are used, these adapters are often wrapped by a `SourcePollingChannelAdapter`. For example, when retrieving messages from a remote FTP Server location, the adapter described in FTP Inbound Channel Adapter is configured with a poller to periodically retrieve messages. So, when components are configured with pollers, the resulting instances are of one of the following types:

- `PollingConsumer`
- `SourcePollingChannelAdapter`

This means that pollers are used in both inbound and outbound messaging scenarios. Here are some use cases in which pollers are used:

- Polling certain external systems, such as FTP Servers, Databases, and Web Services
- Polling internal (pollable) message channels
- Polling internal services (such as repeatedly executing methods on a Java class)

AOP advice classes can be applied to pollers, in an `advice-chain`, such as a transaction advice to start a transaction. Starting with version 4.1, a `PollSkipAdvice` is provided. Pollers use triggers to determine the time of the next poll. The `PollSkipAdvice` can be used to suppress (skip) a poll, perhaps because there is some downstream condition that would prevent the message being processed. To use this advice, you have to provide it with an implementation of a `PollSkipStrategy`. Starting with version 4.2.5, a `SimplePollSkipStrategy` is provided. To use it, you can add an instance as a bean to the application context, inject it into a `PollSkipAdvice`, and add that to the poller's advice chain. To skip polling, call `skipPolls()`. To resume polling, call `reset()`. Version 4.2 added more flexibility in this area. See Conditional Pollers for Message Sources.

This chapter is meant to only give a high-level overview of polling consumers and how they fit into the concept of message channels (see Message Channels) and channel adapters (see Channel Adapter). For more information regarding messaging endpoints in general and polling consumers in particular, see Message Endpoints.

6.2.3. Deferred Acknowledgment Pollable Message Source

Starting with version 5.0.1, certain modules provide `MessageSource` implementations that support deferring acknowledgment until the downstream flow completes (or hands off the message to another thread). This is currently limited to the `AmqpMessageSource` and the `KafkaMessageSource` provided by the `spring-integration-kafka` extension project.

With these message sources, the `IntegrationMessageHeaderAccessor.ACKNOWLEDGMENT_CALLBACK` header (see `MessageHeaderAccessor` API) is added to the message. When used with pollable message sources, the value of the header is an instance of `AcknowledgmentCallback`, as the following example shows:

```
@FunctionalInterface
public interface AcknowledgmentCallback {

    void acknowledge(Status status);

    boolean isAcknowledged();

    void noAutoAck();

    default boolean isAutoAck();

    enum Status {

        /**
         * Mark the message as accepted.
         */
        ACCEPT,

        /**
         * Mark the message as rejected.
         */
        REJECT,

        /**
         * Reject the message and requeue so that it will be redelivered.
         */
        REQUEUE

    }

}
```

Not all message sources (for example, Kafka) support the `REJECT` status. It is treated the same as `ACCEPT`.

Applications can acknowledge a message at any time, as the following example shows:

```
Message<?> received = source.receive();

...

StaticMessageHeaderAccessor.getAcknowledgmentCallback(received)
        .acknowledge(Status.ACCEPT);
```

If the `MessageSource` is wired into a `SourcePollingChannelAdapter`, when the poller thread returns to the adapter after the downstream flow completes, the adapter checks whether the acknowledgment has already been acknowledged and, if not, sets its status to `ACCEPT` it (or `REJECT` if the flow throws an exception). The status values are defined in the `AcknowledgmentCallback.Status` enumeration.

Spring Integration provides `MessageSourcePollingTemplate` to perform ad-hoc polling of a `MessageSource`. This, too, takes care of setting `ACCEPT` or `REJECT` on the `AcknowledgmentCallback` when the `MessageHandler` callback returns (or throws an exception). The following example shows how to poll with the `MessageSourcePollingTemplate`:

```
MessageSourcePollingTemplate template =
    new MessageSourcePollingTemplate(this.source);
template.poll(h -> {
    ...
});
```

In both cases (`SourcePollingChannelAdapter` and `MessageSourcePollingTemplate`), you can disable auto ack/nack by calling `noAutoAck()` on the callback. You might do this if you hand off the message to another thread and wish to acknowledge later. Not all implementations support this (for example, Apache Kafka does not, because the offset commit has to be performed on the same thread).

6.2.4. Conditional Pollers for Message Sources

This section covers how to use conditional pollers.

Background

`Advice` objects, in an `advice-chain` on a poller, advise the whole polling task (both message retrieval and processing). These "around advice" methods do not have access to any context for the poll — only the poll itself. This is fine for requirements such as making a task transactional or skipping a poll due to some external condition, as discussed earlier. What if we wish to take some action depending on the result of the `receive` part of the poll or if we want to adjust the poller depending on conditions? For those instances, Spring Integration offers "Smart" Polling.

"Smart" Polling

Version 5.3 introduced the `ReceiveMessageAdvice` interface. (The `AbstractMessageSourceAdvice` has been deprecated in favor of `default` methods in the `MessageSourceMutator`.) Any `Advice` objects in the `advice-chain` that implement this interface are applied only to the receive operation - `MessageSource.receive()` and `PollableChannel.receive(timeout)`. Therefore they can be applied only for the `SourcePollingChannelAdapter` or `PollingConsumer`. Such classes implement the following methods:

- `beforeReceive(Object source)` This method is called before the `Object.receive()` method. It lets you examine and reconfigure the source. Returning `false` cancels this poll (similar to the `PollSkipAdvice` mentioned earlier).
- `Message<?> afterReceive(Message<?> result, Object source)` This method is called after the `receive()` method. Again, you can reconfigure the source or take any action (perhaps depending on the result, which can be `null` if there was no message created by the source). You can even return a different message

Thread safety

If an advice mutates the the, you should not configure the poller with a `TaskExecutor`. If an advice mutates the source, such mutations are not thread safe and could cause unexpected results, especially with high frequency pollers. If you need to process poll results concurrently, consider using a downstream `ExecutorChannel` instead of adding an executor to the poller.

Advice Chain Ordering

You should understand how the advice chain is processed during initialization. `Advice` objects that do not implement `ReceiveMessageAdvice` are applied to the whole poll process and are all invoked first, in order, before any `ReceiveMessageAdvice`. Then `ReceiveMessageAdvice` objects are invoked in order around the source `receive()` method. If you have, for example, `Advice` objects `a, b, c, d`, where `b` and `d` are `ReceiveMessageAdvice`, the objects are applied in the following order: `a, c, b, d`. Also, if a source is already a `Proxy`, the `ReceiveMessageAdvice` is invoked after any existing `Advice` objects. If you wish to change the order, you must wire up the proxy yourself.

`SimpleActiveIdleReceiveMessageAdvice`

(The previous `SimpleActiveIdleMessageSourceAdvice` for only `MessageSource` is deprecated.) This advice is a simple implementation of `ReceiveMessageAdvice`. When used in conjunction with a `DynamicPeriodicTrigger`, it adjusts the polling frequency, depending on whether or not the previous poll resulted in a message or not. The poller must also have a reference to the same `DynamicPeriodicTrigger`.

Important: Async Handoff

`SimpleActiveIdleReceiveMessageAdvice` modifies the trigger based on the `receive()` result. This works only if the advice is called on the poller thread. It does not work if the poller has a `task-executor`. To use this advice where you wish to use async operations after the result of a poll, do the async handoff later, perhaps by using an `ExecutorChannel`.

`CompoundTriggerAdvice`

This advice allows the selection of one of two triggers based on whether a poll returns a message or not. Consider a poller that uses a `CronTrigger`. `CronTrigger` instances are immutable, so they cannot be altered once constructed. Consider a use case where we want to use a cron expression to trigger a poll once each hour but, if no message is received, poll once per minute and, when a message is retrieved, revert to using the cron expression.

The advice (and poller) use a `CompoundTrigger` for this purpose. The trigger's `primary` trigger can be a `CronTrigger`. When the advice detects that no message is received, it adds the secondary trigger to the `CompoundTrigger`. When the `CompoundTrigger` instance's `nextExecutionTime` method is invoked, it delegates to the secondary trigger, if present. Otherwise, it delegates to the primary trigger.

The poller must also have a reference to the same `CompoundTrigger`.

The following example shows the configuration for the hourly cron expression with a fallback to every minute:

```
<int:inbound-channel-adapter channel="nullChannel" auto-startup="false">
    <bean class="org.springframework.integration.endpoint.PollerAdviceTests.Source" />
    <int:poller trigger="compoundTrigger">
        <int:advice-chain>
            <bean class="org.springframework.integration.aop.CompoundTriggerAdvice">
                <constructor-arg ref="compoundTrigger"/>
                <constructor-arg ref="secondary"/>
            </bean>
        </int:advice-chain>
    </int:poller>
</int:inbound-channel-adapter>

<bean id="compoundTrigger" class="
org.springframework.integration.util.CompoundTrigger">
    <constructor-arg ref="primary" />
</bean>

<bean id="primary" class="org.springframework.scheduling.support.CronTrigger">
    <constructor-arg value="0 0 * * * *" /> <!-- top of every hour -->
</bean>

<bean id="secondary" class="org.springframework.scheduling.support.PeriodicTrigger">
    <constructor-arg value="60000" />
</bean>
```

Important: Async Handoff

`CompoundTriggerAdvice` modifies the trigger based on the `receive()` result. This works only if the advice is called on the poller thread. It does not work if the poller has a `task-executor`. To use this advice where you wish to use async operations after the result of a poll, do the async handoff later, perhaps by using an `ExecutorChannel`.

MessageSource-only Advices

Some advices might be applied only for the `MessageSource.receive()` and they don't make sense for `PollableChannel`. For this purpose a `MessageSourceMutator` interface (an extension of the `ReceiveMessageAdvice`) is still present. With `default` methods it fully replaces already deprecated `AbstractMessageSourceAdvice` and should be used in those implementations where only `MessageSource` proxying is expected. See Inbound Channel Adapters: Polling Multiple Servers and Directories for more information.

6.3. Channel Adapter

A channel adapter is a message endpoint that enables connecting a single sender or receiver to a message channel. Spring Integration provides a number of adapters to support various transports,

such as JMS, file, HTTP, web services, mail, and more. Upcoming chapters of this reference guide discuss each adapter. However, this chapter focuses on the simple but flexible method-invoking channel adapter support. There are both inbound and outbound adapters, and each may be configured with XML elements provided in the core namespace. These provide an easy way to extend Spring Integration, as long as you have a method that can be invoked as either a source or a destination.

6.3.1. Configuring An Inbound Channel Adapter

An `inbound-channel-adapter` element can invoke any method on a Spring-managed object and send a non-null return value to a `MessageChannel` after converting the method's output to a `Message`. When the adapter's subscription is activated, a poller tries to receive messages from the source. The poller is scheduled with the `TaskScheduler` according to the provided configuration. To configure the polling interval or cron expression for an individual channel adapter, you can provide a 'poller' element with one of the scheduling attributes, such as 'fixed-rate' or 'cron'. The following example defines two `inbound-channel-adapter` instances:

```
<int:inbound-channel-adapter ref="source1" method="method1" channel="channel1">
    <int:poller fixed-rate="5000"/>
</int:inbound-channel-adapter>

<int:inbound-channel-adapter ref="source2" method="method2" channel="channel2">
    <int:poller cron="30 * 9-17 * * MON-FRI"/>
</int:channel-adapter>
```

See also Channel Adapter Expressions and Scripts.

If no poller is provided, then a single default poller must be registered within the context. See Endpoint Namespace Support for more detail.

Important: Poller Configuration

Some `inbound-channel-adapter` types are backed by a `SourcePollingChannelAdapter`, which means they contain a poller configuration that polls the `MessageSource` (to invoke a custom method that produces the value that becomes a `Message` payload) based on the configuration specified in the Poller. The following example shows the configuration of two pollers:

```
<int:poller max-messages-per-poll="1" fixed-rate="1000"/>

<int:poller max-messages-per-poll="10" fixed-rate="1000"/>
```

In the the first configuration, the polling task is invoked once per poll, and, during each task (poll), the method (which results in the production of the message) is invoked once, based on the `max-messages-per-poll` attribute value. In the second configuration, the polling task is invoked 10 times per poll or until it returns 'null', thus possibly producing ten messages per poll while each poll happens at one-second intervals. However, what happens if the configuration looks like the following example:

```
<int:poller fixed-rate="1000"/>
```

Note that there is no `max-messages-per-poll` specified. As we cover later, the identical poller configuration in the `PollingConsumer` (for example, service-activator, filter, router, and others) would have a default value of -1 for `max-messages-per-poll`, which means “execute the polling task non-stop unless the polling method returns null (perhaps because there are no more messages in the `QueueChannel`)” and then sleep for one second.

However, in the `SourcePollingChannelAdapter`, it is a bit different. The default value for `max-messages-per-poll` is `1`, unless you explicitly set it to a negative value (such as `-1`). This makes sure that the poller can react to lifecycle events (such as start and stop) and prevents it from potentially spinning in an infinite loop if the implementation of the custom method of the `MessageSource` has a potential to never return null and happens to be non-interruptible.

However, if you are sure that your method can return null and you need to poll for as many sources as available per each poll, you should explicitly set `max-messages-per-poll` to a negative value, as the following example shows:

```
<int:poller max-messages-per-poll="-1" fixed-rate="1000"/>
```

6.3.2. Configuring An Outbound Channel Adapter

An `outbound-channel-adapter` element can also connect a `MessageChannel` to any POJO consumer method that should be invoked with the payload of messages sent to that channel. The following example shows how to define an outbound channel adapter:

```
<int:outbound-channel-adapter channel="channel1" ref="target" method="handle"/>

<beans:bean id="target" class="org.MyPojo"/>
```

If the channel being adapted is a `PollableChannel`, you must provide a poller sub-element, as the following example shows:

```
<int:outbound-channel-adapter channel="channel2" ref="target" method="handle">
    <int:poller fixed-rate="3000" />
</int:outbound-channel-adapter>

<beans:bean id="target" class="org.MyPojo"/>
```

You should use a `ref` attribute if the POJO consumer implementation can be reused in other `<outbound-channel-adapter>` definitions. However, if the consumer implementation is referenced by only a single definition of the `<outbound-channel-adapter>`, you can define it as an inner bean, as the following example shows:

```
<int:outbound-channel-adapter channel="channel" method="handle">
    <beans:bean class="org.Foo"/>
</int:outbound-channel-adapter>
```

Using both the `ref` attribute and an inner handler definition in the same `<outbound-channel-adapter>` configuration is not allowed, as it creates an ambiguous condition. Such a configuration results in an exception being thrown.

Any channel adapter can be created without a `channel` reference, in which case it implicitly creates an instance of `DirectChannel`. The created channel's name matches the `id` attribute of the `<inbound-channel-adapter>` or `<outbound-channel-adapter>` element. Therefore, if `channel` is not provided, `id` is required.

6.3.3. Channel Adapter Expressions and Scripts

Like many other Spring Integration components, the `<inbound-channel-adapter>` and `<outbound-channel-adapter>` also provide support for SpEL expression evaluation. To use SpEL, provide the

expression string in the 'expression' attribute instead of providing the 'ref' and 'method' attributes that are used for method-invocation on a bean. When an expression is evaluated, it follows the same contract as method-invocation where: the expression for an `<inbound-channel-adapter>` generates a message any time the evaluation result is a non-null value, while the expression for an `<outbound-channel-adapter>` must be the equivalent of a void-returning method invocation.

Starting with Spring Integration 3.0, an `<int:inbound-channel-adapter/>` can also be configured with a SpEL `<expression/>` (or even with a `<script/>`) sub-element, for when more sophistication is required than can be achieved with the simple 'expression' attribute. If you provide a script as a `Resource` by using the `location` attribute, you can also set `refresh-check-delay`, which allows the resource to be periodically refreshed. If you want the script to be checked on each poll, you would need to coordinate this setting with the poller's trigger, as the following example shows:

```
<int:inbound-channel-adapter ref="source1" method="method1" channel="channel1">
    <int:poller max-messages-per-poll="1" fixed-delay="5000"/>
    <script:script lang="ruby" location="Foo.rb" refresh-check-delay="5000"/>
</int:inbound-channel-adapter>
```

See also the `cacheSeconds` property on the `ReloadableResourceBundleExpressionSource` when using the `<expression/>` sub-element. For more information regarding expressions, see Spring Expression Language (SpEL). For scripts, see Groovy support and Scripting Support.

The `<int:inbound-channel-adapter/>` is endpoint starts a message flow by periodically triggering to poll some underlying `MessageSource`. Since, at the time of polling, there is no message object, expressions and scripts do not have access to a root `Message`, so there are no payload or headers properties that are available in most other messaging SpEL expressions. The script can generate and return a complete `Message` object with headers and payload or only a payload, which is added to a message with basic headers.

6.4. Messaging Bridge

A messaging bridge is a relatively trivial endpoint that connects two message channels or channel adapters. For example, you may want to connect a `PollableChannel` to a `SubscribableChannel` so that the subscribing endpoints do not have to worry about any polling configuration. Instead, the messaging bridge provides the polling configuration.

By providing an intermediary poller between two channels, you can use a messaging bridge to throttle inbound messages. The poller's trigger determines the rate at which messages arrive on the second channel, and the poller's `maxMessagesPerPoll` property enforces a limit on the throughput.

Another valid use for a messaging bridge is to connect two different systems. In such a scenario, Spring Integration's role is limited to making the connection between these systems and managing a poller, if necessary. It is probably more common to have at least a transformer between the two systems, to translate between their formats. In that case, the channels can be provided as the 'input-channel' and 'output-channel' of a transformer endpoint. If data format translation is not required,

the messaging bridge may indeed be sufficient.

6.4.1. Configuring a Bridge with XML

You can use the `<bridge>` element is used to create a messaging bridge between two message channels or channel adapters. To do so, provide the `input-channel` and `output-channel` attributes, as the following example shows:

```
<int:bridge input-channel="input" output-channel="output"/>
```

As mentioned above, a common use case for the messaging bridge is to connect a `PollableChannel` to a `SubscribableChannel`. When performing this role, the messaging bridge may also serve as a throttler:

```
<int:bridge input-channel="pollable" output-channel="subscribable">
    <int:poller max-messages-per-poll="10" fixed-rate="5000"/>
 </int:bridge>
```

You can use a similar mechanism to connecting channel adapters. The following example shows a simple “echo” between the `stdin` and `stdout` adapters from Spring Integration’s `stream` namespace:

```
<int-stream:stdin-channel-adapter id="stdin"/>

 <int-stream:stdout-channel-adapter id="stdout"/>

 <int:bridge id="echo" input-channel="stdin" output-channel="stdout"/>
```

Similar configurations work for other (potentially more useful) Channel Adapter bridges, such as file-to-JMS or mail-to-file. Upcoming chapters cover the various channel adapters.

If no 'output-channel' is defined on a bridge, the reply channel provided by the inbound message is used, if available. If neither an output nor a reply channel is available, an exception is thrown.

6.4.2. Configuring a Bridge with Java Configuration

The following example shows how to configure a bridge in Java by using the `@BridgeFrom` annotation:

```
@Bean
public PollableChannel polled() {
    return new QueueChannel();
}

@Bean
@BridgeFrom(value = "polled", poller = @Poller(fixedDelay = "5000",
maxMessagesPerPoll = "10"))
public SubscribableChannel direct() {
    return new DirectChannel();
}
```

The following example shows how to configure a bridge in Java by using the `@BridgeTo` annotation:

```
@Bean
@BridgeTo(value = "direct", poller = @Poller(fixedDelay = "5000", maxMessagesPerPoll =
"10"))
public PollableChannel polled() {
    return new QueueChannel();
}

@Bean
public SubscribableChannel direct() {
    return new DirectChannel();
}
```

Alternately, you can use a `BridgeHandler`, as the following example shows:

```
@Bean
@ServiceActivator(inputChannel = "polled",
        poller = @Poller(fixedRate = "5000", maxMessagesPerPoll = "10"))
public BridgeHandler bridge() {
    BridgeHandler bridge = new BridgeHandler();
    bridge.setOutputChannelName("direct");
    return bridge;
}
```

6.4.3. Configuring a Bridge with the Java DSL

You can use the Java Domain Specific Language (DSL) to configure a bridge, as the following example shows:

```
@Bean
public IntegrationFlow bridgeFlow() {
    return IntegrationFlows.from("polled")
            .bridge(e -> e.poller(Pollers.fixedDelay(5000).maxMessagesPerPoll(10)))
            .channel("direct")
            .get();
}
```

Chapter 7. Message

The Spring Integration `Message` is a generic container for data. Any object can be provided as the payload, and each `Message` instance includes headers containing user-extensible properties as key-value pairs.

7.1. The `Message` Interface

The following listing shows the definition of the `Message` interface:

```
public interface Message<T> {

    T getPayload();

    MessageHeaders getHeaders();

}
```

The `Message` interface is a core part of the API. By encapsulating the data in a generic wrapper, the messaging system can pass it around without any knowledge of the data's type. As an application evolves to support new types or when the types themselves are modified or extended, the messaging system is not affected. On the other hand, when some component in the messaging system does require access to information about the `Message`, such metadata can typically be stored to and retrieved from the metadata in the message headers.

7.2. Message Headers

Just as Spring Integration lets any `Object` be used as the payload of a `Message`, it also supports any `Object` types as header values. In fact, the `MessageHeaders` class implements the `java.util.Map_interface`, as the following class definition shows:

```
public final class MessageHeaders implements Map<String, Object>, Serializable {
  ...
}
```

Even though the `MessageHeaders` class implements `Map`, it is effectively a read-only implementation. Any attempt to `put` a value in the Map results in an `UnsupportedOperationException`. The same applies for `remove` and `clear`. Since messages may be passed to multiple consumers, the structure of the `Map` cannot be modified. Likewise, the message's payload `Object` can not be `set` after the initial creation. However, the mutability of the header values themselves (or the payload Object) is intentionally left as a decision for the framework user.

As an implementation of `Map`, the headers can be retrieved by calling `get(..)` with the name of the header. Alternatively, you can provide the expected `Class` as an additional parameter. Even better, when retrieving one of the pre-defined values, convenient getters are available. The following example shows each of these three options:

```
Object someValue = message.getHeaders().get("someKey");

CustomerId customerId = message.getHeaders().get("customerId", CustomerId.class);

Long timestamp = message.getHeaders().getTimestamp();
```

The following table describes the pre-defined message headers:

Table 1. Pre-defined Message Headers

Header Name	Header Type	Usage
`MessageHeaders.ID`	`java.util.UUID`	An identifier for this message instance. Changes each time a message is mutated.
`MessageHeaders.TIMESTAMP`	`java.lang.Long`	The time the message was created. Changes each time a message is mutated.
`MessageHeaders.REPLY_CHANNEL`	`java.lang.Object (String or MessageChannel)`	A channel to which a reply (if any) is sent when no explicit output channel is configured and there is no `ROUTING_SLIP` or the `ROUTING_SLIP` is exhausted. If the value is a `String`, it must represent a bean name or have been generated by a `ChannelRegistry`.
`MessageHeaders.ERROR_CHANNEL`	`java.lang.Object (String or MessageChannel)`	A channel to which errors are sent. If the value is a `String`, it must represent a bean name or have been generated by a `ChannelRegistry`.

Many inbound and outbound adapter implementations also provide or expect certain headers, and you can configure additional user-defined headers. Constants for these headers can be found in those modules where such headers exist — for example. `AmqpHeaders`, `JmsHeaders`, and so on.

7.2.1. `MessageHeaderAccessor` API

Starting with Spring Framework 4.0 and Spring Integration 4.0, the core messaging abstraction has been moved to the `spring-messaging` module, and the `MessageHeaderAccessor` API has been introduced to provide additional abstraction over messaging implementations. All (core) Spring Integration-specific message headers constants are now declared in the `IntegrationMessageHeaderAccessor` class. The following table describes the pre-defined message headers:

Table 2. Pre-defined Message Headers

Header Name	Header Type	Usage
IntegrationMessageHeaderAccessor.CORRELATION_ID	java.lang.Object	Used to correlate two or more messages.
IntegrationMessageHeaderAccessor.SEQUENCE_NUMBER	java.lang.Integer	Usually a sequence number with a group of messages with a SEQUENCE_SIZE but can also be used in a <resequencer/> to resequence an unbounded group of messages.
IntegrationMessageHeaderAccessor.SEQUENCE_SIZE	java.lang.Integer	The number of messages within a group of correlated messages.
IntegrationMessageHeaderAccessor.EXPIRATION_DATE	java.lang.Long	Indicates when a message is expired. Not used by the framework directly but can be set with a header enricher and used in a <filter/> that is configured with an UnexpiredMessageSelector.
IntegrationMessageHeaderAccessor.PRIORITY	java.lang.Integer	Message priority — for example, within a PriorityChannel.
IntegrationMessageHeaderAccessor.DUPLICATE_MESSAGE	java.lang.Boolean	True if a message was detected as a duplicate by an idempotent receiver interceptor. See Idempotent Receiver Enterprise Integration Pattern.
IntegrationMessageHeaderAccessor.CLOSEABLE_RESOURCE	java.io.Closeable	This header is present if the message is associated with a Closeable that should be closed when message processing is complete. An example is the Session associated with a streamed file transfer using FTP, SFTP, and so on.
IntegrationMessageHeaderAccessor.DELIVERY_ATTEMPT	java.lang.AtomicInteger	If a message-driven channel adapter supports the configuration of a RetryTemplate, this header contains the current delivery attempt.
IntegrationMessageHeaderAccessor.ACKNOWLEDGMENT_CALLBACK	o.s.i.support.AcknowledgmentCallback	If an inbound endpoint supports it, a call back to accept, reject, or requeue a message. See Deferred Acknowledgment Pollable Message Source and MQTT Manual Acks.

Convenient typed getters for some of these headers are provided on the IntegrationMessageHeaderAccessor class, as the following example shows:

```
IntegrationMessageHeaderAccessor accessor = new IntegrationMessageHeaderAccessor
(message);
int sequenceNumber = accessor.getSequenceNumber();
Object correlationId = accessor.getCorrelationId();
...
```

The following table describes headers that also appear in the `IntegrationMessageHeaderAccessor` but are generally not used by user code (that is, they are generally used by internal parts of Spring Integration — their inclusion here is for completeness):

Table 3. Pre-defined Message Headers

Header Name	Header Type	Usage
`IntegrationMessageHeaderAccessor.SEQUENCE_DETAILS`	`java.util.List<List<Object>>`	A stack of correlation data used when nested correlation is needed (for example, `splitter→⋯→splitter→⋯→aggregator→⋯→aggregator`).
`IntegrationMessageHeaderAccessor.ROUTING_SLIP`	`java.util.Map<List<Object>, Integer>`	See Routing Slip.

7.2.2. Message ID Generation

When a message transitions through an application, each time it is mutated (for example, by a transformer) a new message ID is assigned. The message ID is a `UUID`. Beginning with Spring Integration 3.0, the default strategy used for IS generation is more efficient than the previous `java.util.UUID.randomUUID()` implementation. It uses simple random numbers based on a secure random seed instead of creating a secure random number each time.

A different UUID generation strategy can be selected by declaring a bean that implements `org.springframework.util.IdGenerator` in the application context.

Only one UUID generation strategy can be used in a classloader. This means that, if two or more application contexts run in the same classloader, they share the same strategy. If one of the contexts changes the strategy, it is used by all contexts. If two or more contexts in the same classloader declare a bean of type `org.springframework.util.IdGenerator`, they must all be an instance of the same class. Otherwise, the context attempting to replace a custom strategy fails to initialize. If the strategy is the same, but parameterized, the strategy in the first context to be initialized is used.

In addition to the default strategy, two additional `IdGenerators` are provided. `org.springframework.util.JdkIdGenerator` uses the previous `UUID.randomUUID()` mechanism. You can use `o.s.i.support.IdGenerators.SimpleIncrementingIdGenerator` when a UUID is not really needed and a simple incrementing value is sufficient.

7.2.3. Read-only Headers

The `MessageHeaders.ID` and `MessageHeaders.TIMESTAMP` are read-only headers and cannot be overridden.

Since version 4.3.2, the `MessageBuilder` provides the `readOnlyHeaders(String… readOnlyHeaders)` API to customize a list of headers that should not be copied from an upstream `Message`. Only the `MessageHeaders.ID` and `MessageHeaders.TIMESTAMP` are read only by default. The global `spring.integration.readOnly.headers` property (see Global Properties) is provided to customize `DefaultMessageBuilderFactory` for framework components. This can be useful when you would like do not populate some out-of-the-box headers, such as `contentType` by the `ObjectToJsonTransformer` (see JSON Transformers).

When you try to build a new message using `MessageBuilder`, this kind of header is ignored and a particular `INFO` message is emitted to logs.

Starting with version 5.0, Messaging Gateway, Header Enricher, Content Enricher and Header Filter do not let you configure the `MessageHeaders.ID` and `MessageHeaders.TIMESTAMP` header names when `DefaultMessageBuilderFactory` is used, and they throw `BeanInitializationException`.

7.2.4. Header Propagation

When messages are processed (and modified) by message-producing endpoints (such as a service activator), in general, inbound headers are propagated to the outbound message. One exception to this is a transformer, when a complete message is returned to the framework. In that case, the user code is responsible for the entire outbound message. When a transformer just returns the payload, the inbound headers are propagated. Also, a header is only propagated if it does not already exist in the outbound message, letting you change header values as needed.

Starting with version 4.3.10, you can configure message handlers (that modify messages and produce output) to suppress the propagation of specific headers. To configure the header(s) you do not want to be copied, call the `setNotPropagatedHeaders()` or `addNotPropagatedHeaders()` methods on the `MessageProducingMessageHandler` abstract class.

You can also globally suppress propagation of specific message headers by setting the `readOnlyHeaders` property in `META-INF/spring.integration.properties` to a comma-delimited list of headers.

Starting with version 5.0, the `setNotPropagatedHeaders()` implementation on the `AbstractMessageProducingHandler` applies simple patterns (`xxx*`, `xxx`, `*xxx`, or `xxx*yyy`) to allow filtering headers with a common suffix or prefix. See `PatternMatchUtils` Javadoc for more information. When one of the patterns is `*` (asterisk), no headers are propagated. All other patterns are ignored. In that case, the service activator behaves the same way as a transformer and any required headers must be supplied in the `Message` returned from the service method. The `notPropagatedHeaders()` option is available in the `ConsumerEndpointSpec` for the Java DSL It is also available for XML configuration of the `<service-activator>` component as a `not-propagated-headers` attribute.

Header propagation suppression does not apply to those endpoints that do not modify the message, such as bridges and routers.

7.3. Message Implementations

The base implementation of the `Message` interface is `GenericMessage<T>`, and it provides two constructors, shown in the following listing:

```
new GenericMessage<T>(T payload);

new GenericMessage<T>(T payload, Map<String, Object> headers)
```

When a `Message` is created, a random unique ID is generated. The constructor that accepts a `Map` of headers copies the provided headers to the newly created `Message`.

There is also a convenient implementation of `Message` designed to communicate error conditions. This implementation takes a `Throwable` object as its payload, as the following example shows:

```
ErrorMessage message = new ErrorMessage(someThrowable);

Throwable t = message.getPayload();
```

Note that this implementation takes advantage of the fact that the `GenericMessage` base class is parameterized. Therefore, as shown in both examples, no casting is necessary when retrieving the `Message` payload `Object`.

7.4. The `MessageBuilder` Helper Class

You may notice that the `Message` interface defines retrieval methods for its payload and headers but provides no setters. The reason for this is that a `Message` cannot be modified after its initial creation. Therefore, when a `Message` instance is sent to multiple consumers (for example, through a publish-subscribe Channel), if one of those consumers needs to send a reply with a different payload type, it must create a new `Message`. As a result, the other consumers are not affected by those changes. Keep in mind that multiple consumers may access the same payload instance or header value, and whether such an instance is itself immutable is a decision left to you. In other words, the contract for `Message` instances is similar to that of an unmodifiable `Collection`, and the `MessageHeaders` map further exemplifies that. Even though the `MessageHeaders` class implements `java.util.Map`, any attempt to invoke a `put` operation (or 'remove' or 'clear') on a `MessageHeaders` instance results in an `UnsupportedOperationException`.

Rather than requiring the creation and population of a Map to pass into the GenericMessage constructor, Spring Integration does provide a far more convenient way to construct Messages:

`MessageBuilder`. The `MessageBuilder` provides two factory methods for creating `Message` instances from either an existing `Message` or with a payload `Object`. When building from an existing `Message`, the headers and payload of that `Message` are copied to the new `Message`, as the following example shows:

```
Message<String> message1 = MessageBuilder.withPayload("test")
        .setHeader("foo", "bar")
        .build();

Message<String> message2 = MessageBuilder.fromMessage(message1).build();

assertEquals("test", message2.getPayload());
assertEquals("bar", message2.getHeaders().get("foo"));
```

If you need to create a `Message` with a new payload but still want to copy the headers from an existing `Message`, you can use one of the 'copy' methods, as the following example shows:

```
Message<String> message3 = MessageBuilder.withPayload("test3")
        .copyHeaders(message1.getHeaders())
        .build();

Message<String> message4 = MessageBuilder.withPayload("test4")
        .setHeader("foo", 123)
        .copyHeadersIfAbsent(message1.getHeaders())
        .build();

assertEquals("bar", message3.getHeaders().get("foo"));
assertEquals(123, message4.getHeaders().get("foo"));
```

Note that the `copyHeadersIfAbsent` method does not overwrite existing values. Also, in the preceding example, you can see how to set any user-defined header with `setHeader`. Finally, there are `set` methods available for the predefined headers as well as a non-destructive method for setting any header (`MessageHeaders` also defines constants for the pre-defined header names).

You can also use `MessageBuilder` to set the priority of messages, as the following example shows:

```
Message<Integer> importantMessage = MessageBuilder.withPayload(99)
        .setPriority(5)
        .build();

assertEquals(5, importantMessage.getHeaders().getPriority());

Message<Integer> lessImportantMessage = MessageBuilder.fromMessage(importantMessage)
        .setHeaderIfAbsent(IntegrationMessageHeaderAccessor.PRIORITY, 2)
        .build();

assertEquals(2, lessImportantMessage.getHeaders().getPriority());
```

The `priority` header is considered only when using a `PriorityChannel` (as described in the next chapter). It is defined as a `java.lang.Integer`.

Chapter 8. Message Routing

This chapter covers the details of using Spring Integration to route messages.

8.1. Routers

This section covers how routers work. It includes the following topics:

- Overview
- Common Router Parameters
- Router Implementations
- Configuring a Generic Router
- Routers and the Spring Expression Language (SpEL)
- Dynamic Routers

8.1.1. Overview

Routers are a crucial element in many messaging architectures. They consume messages from a message channel and forward each consumed message to one or more different message channels depending on a set of conditions.

Spring Integration provides the following routers:

- Payload Type Router
- Header Value Router
- Recipient List Router
- XPath Router (part of the XML module)
- Error Message Exception Type Router
- (Generic) Router

Router implementations share many configuration parameters. However, certain differences exist between routers. Furthermore, the availability of configuration parameters depends on whether routers are used inside or outside of a chain. In order to provide a quick overview, all available attributes are listed in the two following tables .

The following table shows the configuration parameters available for a router outside of a chain:

Table 4. Routers Outside of a Chain

Attribute	router	header value router	xpath router	payload type router	recipient list route	exception type router
apply-sequence	✓	✓	✓	✓	✓	✓

Attribute	router	header value router	xpath router	payload type router	recipient list route	exception type router
default-output-channel	✓	✓	✓	✓	✓	✓
resolution-required	✓	✓	✓	✓	✓	✓
ignore-send-failures	✓	✓	✓	✓	✓	✓
timeout	✓	✓	✓	✓	✓	✓
id	✓	✓	✓	✓	✓	✓
auto-startup	✓	✓	✓	✓	✓	✓
input-channel	✓	✓	✓	✓	✓	✓
order	✓	✓	✓	✓	✓	✓
method	✓					
ref	✓					
expression	✓					
header-name		✓				
evaluate-as-string			✓			
xpath-expression-ref			✓			
converter			✓			

The following table shows the configuration parameters available for a router inside of a chain:

Table 5. Routers Inside of a Chain

Attribute	router	header value router	xpath router	payload type router	recipient list router	exception type router
apply-sequence	✓	✓	✓	✓	✓	✓
default-output-channel	✓	✓	✓	✓	✓	✓
resolution-required	✓	✓	✓	✓	✓	✓
ignore-send-failures	✓	✓	✓	✓	✓	✓
timeout	✓	✓	✓	✓	✓	✓
id						
auto-startup						
input-channel						
order						
method	✓					
ref	✓					
expression	✓					
header-name		✓				
evaluate-as-string			✓			
xpath-expression-ref			✓			
converter			✓			

As of Spring Integration 2.1, router parameters have been more standardized across all router implementations. Consequently, a few minor changes may break older Spring Integration based applications.

Since Spring Integration 2.1, the `ignore-channel-name-resolution-failures` attribute is removed in favor of consolidating its behavior with the `resolution-required` attribute. Also, the `resolution-required` attribute now defaults to `true`.

Prior to these changes, the `resolution-required` attribute defaulted to `false`, causing messages to be silently dropped when no channel was resolved and no `default-output-channel` was set. The new behavior requires at least one resolved channel and, by default, throws a `MessageDeliveryException` if no channel was determined (or an attempt to send was not successful).

If you do desire to drop messages silently, you can set `default-output-channel="nullChannel"`.

8.1.2. Common Router Parameters

This section describes the parameters common to all router parameters (the parameters with all their boxes ticked in the two tables shown earlier in this chapter).

Inside and Outside of a Chain

The following parameters are valid for all routers inside and outside of chains.

`apply-sequence`

This attribute specifies whether sequence number and size headers should be added to each message. This optional attribute defaults to `false`.

`default-output-channel`

If set, this attribute provides a reference to the channel where messages should be sent if channel resolution fails to return any channels. If no default output channel is provided, the router throws an exception. If you would like to silently drop those messages instead, set the default output channel attribute value to `nullChannel`.

A message is sent only to the `default-output-channel` if `resolution-required` is `false` and the channel is not resolved.

`resolution-required`

This attribute specifies whether channel names must always be successfully resolved to channel instances that exist. If set to `true`, a `MessagingException` is raised when the channel cannot be resolved. Setting this attribute to `false` causes any unresovable channels to be ignored. This optional attribute defaults to `true`.

A Message is sent only to the `default-output-channel`, if specified, when `resolution-required` is `false` and the channel is not resolved.

`ignore-send-failures`

If set to `true`, failures to send to a message channel is ignored. If set to `false`, a `MessageDeliveryException` is thrown instead, and, if the router resolves more than one channel, any subsequent channels do not receive the message.

The exact behavior of this attribute depends on the type of the `Channel` to which the messages are sent. For example, when using direct channels (single threaded), send failures can be caused by exceptions thrown by components much further downstream. However, when sending messages to a simple queue channel (asynchronous), the likelihood of an exception to be thrown is rather remote.

While most routers route to a single channel, they can return more than one channel name. The `recipient-list-router`, for instance, does exactly that. If you set this attribute to `true` on a router that only routes to a single channel, any caused exception is swallowed, which usually makes little sense. In that case, it would be better to catch the exception in an error flow at the flow entry point. Therefore, setting the `ignore-send-failures` attribute to `true` usually makes more sense when the router implementation returns more than one channel name, because the other channel(s) following the one that fails would still receive the message.

This attribute defaults to `false`.

`timeout`

The `timeout` attribute specifies the maximum amount of time in milliseconds to wait when sending messages to the target Message Channels. By default, the send operation blocks indefinitely.

Top-Level (Outside of a Chain)

The following parameters are valid only across all top-level routers that are outside of chains.

`id`

Identifies the underlying Spring bean definition, which, in the case of routers, is an instance of `EventDrivenConsumer` or `PollingConsumer`, depending on whether the router's `input-channel` is a `SubscribableChannel` or a `PollableChannel`, respectively. This is an optional attribute.

`auto-startup`

This "lifecycle" attribute signaled whether this component should be started during startup of the application context. This optional attribute defaults to `true`.

`input-channel`

The receiving message channel of this endpoint.

`order`

This attribute defines the order for invocation when this endpoint is connected as a subscriber to a channel. This is particularly relevant when that channel uses a failover dispatching strategy. It has no effect when this endpoint itself is a polling consumer for a channel with a queue.

8.1.3. Router Implementations

Since content-based routing often requires some domain-specific logic, most use cases require Spring Integration's options for delegating to POJOs by using either the XML namespace support or annotations. Both of these are discussed later. However, we first present a couple of implementations that fulfill common requirements.

`PayloadTypeRouter`

A `PayloadTypeRouter` sends messages to the channel defined by payload-type mappings, as the following example shows:

```
<bean id="payloadTypeRouter"
      class="org.springframework.integration.router.PayloadTypeRouter">
    <property name="channelMapping">
        <map>
            <entry key="java.lang.String" value-ref="stringChannel"/>
            <entry key="java.lang.Integer" value-ref="integerChannel"/>
        </map>
    </property>
</bean>
```

Configuration of the `PayloadTypeRouter` is also supported by the namespace provided by Spring Integration (see `Namespace Support`), which essentially simplifies configuration by combining the `<router/>` configuration and its corresponding implementation (defined by using a `<bean/>` element) into a single and more concise configuration element. The following example shows a `PayloadTypeRouter` configuration that is equivalent to the one above but uses the namespace support:

```
<int:payload-type-router input-channel="routingChannel">
    <int:mapping type="java.lang.String" channel="stringChannel" />
    <int:mapping type="java.lang.Integer" channel="integerChannel" />
</int:payload-type-router>
```

The following example shows the equivalent router configured in Java:

```
@ServiceActivator(inputChannel = "routingChannel")
@Bean
public PayloadTypeRouter router() {
    PayloadTypeRouter router = new PayloadTypeRouter();
    router.setChannelMapping(String.class.getName(), "stringChannel");
    router.setChannelMapping(Integer.class.getName(), "integerChannel");
    return router;
}
```

When using the Java DSL, there are two options.

First, you can define the router object as shown in the preceding example:

```
@Bean
public IntegrationFlow routerFlow1() {
    return IntegrationFlows.from("routingChannel")
            .route(router())
            .get();
}

public PayloadTypeRouter router() {
    PayloadTypeRouter router = new PayloadTypeRouter();
    router.setChannelMapping(String.class.getName(), "stringChannel");
    router.setChannelMapping(Integer.class.getName(), "integerChannel");
    return router;
}
```

Note that the router can be, but does not have to be, a `@Bean`. The flow registers it if it is not a `@Bean`.

Second, you can define the routing function within the DSL flow itself, as the following example shows:

```
@Bean
public IntegrationFlow routerFlow2() {
    return IntegrationFlows.from("routingChannel")
            .<Object, Class<?>>route(Object::getClass, m -> m
                    .channelMapping(String.class, "stringChannel")
                    .channelMapping(Integer.class, "integerChannel"))
            .get();
}
```

`HeaderValueRouter`

A `HeaderValueRouter` sends Messages to the channel based on the individual header value mappings.

When a `HeaderValueRouter` is created, it is initialized with the name of the header to be evaluated. The value of the header could be one of two things:

- An arbitrary value
- A channel name

If it is an arbitrary value, additional mappings for these header values to channel names are required. Otherwise, no additional configuration is needed.

Spring Integration provides a simple namespace-based XML configuration to configure a `HeaderValueRouter`. The following example demonstrates configuration for the `HeaderValueRouter` when mapping of header values to channels is required:

```
<int:header-value-router input-channel="routingChannel" header-name="testHeader">
    <int:mapping value="someHeaderValue" channel="channelA" />
    <int:mapping value="someOtherHeaderValue" channel="channelB" />
</int:header-value-router>
```

During the resolution process, the router defined in the preceding example may encounter channel resolution failures, causing an exception. If you want to suppress such exceptions and send unresolved messages to the default output channel (identified with the `default-output-channel` attribute) set `resolution-required` to `false`.

Normally, messages for which the header value is not explicitly mapped to a channel are sent to the `default-output-channel`. However, when the header value is mapped to a channel name but the channel cannot be resolved, setting the `resolution-required` attribute to `false` results in routing such messages to the `default-output-channel`.

As of Spring Integration 2.1, the attribute was changed from `ignore-channel-name-resolution-failures` to `resolution-required`. Attribute `resolution-required` defaults to `true`.

The following example shows the equivalent router configured in Java:

```
@ServiceActivator(inputChannel = "routingChannel")
@Bean
public HeaderValueRouter router() {
    HeaderValueRouter router = new HeaderValueRouter("testHeader");
    router.setChannelMapping("someHeaderValue", "channelA");
    router.setChannelMapping("someOtherHeaderValue", "channelB");
    return router;
}
```

When using the Java DSL, there are two options. First, you can define the router object as shown in

the preceding example:

```
@Bean
public IntegrationFlow routerFlow1() {
    return IntegrationFlows.from("routingChannel")
            .route(router())
            .get();
}

public HeaderValueRouter router() {
    HeaderValueRouter router = new HeaderValueRouter("testHeader");
    router.setChannelMapping("someHeaderValue", "channelA");
    router.setChannelMapping("someOtherHeaderValue", "channelB");
    return router;
}
```

Note that the router can be, but does not have to be, a `@Bean`. The flow registers it if it is not a `@Bean`.

Second, you can define the routing function within the DSL flow itself, as the following example shows:

```
@Bean
public IntegrationFlow routerFlow2() {
    return IntegrationFlows.from("routingChannel")
            .<Message<?>, String>route(m -> m.getHeaders().get("testHeader",
String.class), m -> m
                    .channelMapping("someHeaderValue", "channelA")
                    .channelMapping("someOtherHeaderValue", "channelB"),
                e -> e.id("headerValueRouter"))
            .get();
}
```

Configuration where mapping of header values to channel names is not required, because header values themselves represent channel names. The following example shows a router that does not require mapping of header values to channel names:

```
<int:header-value-router input-channel="routingChannel" header-name="testHeader"/>
```

Since Spring Integration 2.1, the behavior of resolving channels is more explicit. For example, if you omit the `default-output-channel` attribute, the router was unable to resolve at least one valid channel, and any channel name resolution failures were ignored by setting `resolution-required` to `false`, then a `MessageDeliveryException` is thrown.

Basically, by default, the router must be able to route messages successfully to at least one channel. If you really want to drop messages, you must also have `default-output-channel` set to `nullChannel`.

`RecipientListRouter`

A `RecipientListRouter` sends each received message to a statically defined list of message channels. The following example creates a `RecipientListRouter`:

```
<bean id="recipientListRouter"
      class="org.springframework.integration.router.RecipientListRouter">
    <property name="channels">
        <list>
            <ref bean="channel1"/>
            <ref bean="channel2"/>
            <ref bean="channel3"/>
        </list>
    </property>
</bean>
```

Spring Integration also provides namespace support for the `RecipientListRouter` configuration (see Namespace Support) as the following example shows:

```
<int:recipient-list-router id="customRouter" input-channel="routingChannel"
        timeout="1234"
        ignore-send-failures="true"
        apply-sequence="true">
  <int:recipient channel="channel1"/>
  <int:recipient channel="channel2"/>
</int:recipient-list-router>
```

The following example shows the equivalent router configured in Java:

```
@ServiceActivator(inputChannel = "routingChannel")
@Bean
public RecipientListRouter router() {
    RecipientListRouter router = new RecipientListRouter();
    router.setSendTimeout(1_234L);
    router.setIgnoreSendFailures(true);
    router.setApplySequence(true);
    router.addRecipient("channel1");
    router.addRecipient("channel2");
    router.addRecipient("channel3");
    return router;
}
```

The following example shows the equivalent router configured by using the Java DSL:

```
@Bean
public IntegrationFlow routerFlow() {
    return IntegrationFlows.from("routingChannel")
            .routeToRecipients(r -> r
                    .applySequence(true)
                    .ignoreSendFailures(true)
                    .recipient("channel1")
                    .recipient("channel2")
                    .recipient("channel3")
                    .sendTimeout(1_234L))
            .get();
}
```

The 'apply-sequence' flag here has the same effect as it does for a publish-subscribe-channel, and, as with a publish-subscribe-channel, it is disabled by default on the `recipient-list-router`. See `PublishSubscribeChannel` Configuration for more information.

Another convenient option when configuring a `RecipientListRouter` is to use Spring Expression Language (SpEL) support as selectors for individual recipient channels. Doing so is similar to using a filter at the beginning of a 'chain' to act as a “selective consumer”. However, in this case, it is all combined rather concisely into the router’s configuration, as the following example shows:

```
<int:recipient-list-router id="customRouter" input-channel="routingChannel">
    <int:recipient channel="channel1" selector-expression="payload.equals('foo')"/>
    <int:recipient channel="channel2" selector-expression="headers.containsKey('bar')
"/>
</int:recipient-list-router>
```

In the preceding configuration, a SpEL expression identified by the `selector-expression` attribute is evaluated to determine whether this recipient should be included in the recipient list for a given input message. The evaluation result of the expression must be a boolean. If this attribute is not defined, the channel is always among the list of recipients.

`RecipientListRouterManagement`

Starting with version 4.1, the `RecipientListRouter` provides several operations to manipulate recipients dynamically at runtime. These management operations are presented by `RecipientListRouterManagement` through the `@ManagedResource` annotation. They are available by using Control Bus as well as by using JMX, as the following example shows:

```
<control-bus input-channel="controlBus"/>

<recipient-list-router id="simpleRouter" input-channel="routingChannelA">
   <recipient channel="channel1"/>
</recipient-list-router>

<channel id="channel2"/>
```

```
messagingTemplate.convertAndSend(controlBus,
"@'simpleRouter.handler'.addRecipient('channel2')");
```

From the application start up the `simpleRouter`, has only one `channel1` recipient. But after the `addRecipient` command, `channel2` recipient is added. It is a “registering an interest in something that is part of the message” use case, when we may be interested in messages from the router at some time period, so we are subscribing to the the `recipient-list-router` and, at some point, decide to unsubscribe.

Because of the runtime management operation for the `<recipient-list-router>`, it can be configured without any `<recipient>` from the start. In this case, the behavior of `RecipientListRouter` is the same when there is no one matching recipient for the message. If `defaultOutputChannel` is configured, the message is sent there. Otherwise the `MessageDeliveryException` is thrown.

XPath Router

The XPath Router is part of the XML Module. See Routing XML Messages with XPath.

Routing and Error Handling

Spring Integration also provides a special type-based router called `ErrorMessageExceptionTypeRouter` for routing error messages (defined as messages whose `payload` is a `Throwable` instance). `ErrorMessageExceptionTypeRouter` is similar to the `PayloadTypeRouter`. In fact, they are almost identical. The only difference is that, while `PayloadTypeRouter` navigates the instance hierarchy of a payload instance (for example, `payload.getClass().getSuperclass()`) to find the most specific type and channel mappings, the `ErrorMessageExceptionTypeRouter` navigates the hierarchy of 'exception

causes' (for example, `payload.getCause()`) to find the most specific `Throwable` type or channel mappings and uses `mappingClass.isInstance(cause)` to match the `cause` to the class or any super class.

The channel mapping order in this case matters. So, if there is a requirement to get mapping for an `IllegalArgumentException`, but not a `RuntimeException`, the last one must be configured on router first.

Since version 4.3 the `ErrorMessageExceptionTypeRouter` loads all mapping classes during the initialization phase to fail-fast for a `ClassNotFoundException`.

The following example shows a sample configuration for `ErrorMessageExceptionTypeRouter`:

```
<int:exception-type-router input-channel="inputChannel"
                           default-output-channel="defaultChannel">
    <int:mapping exception-type="java.lang.IllegalArgumentException"
                 channel="illegalChannel"/>
    <int:mapping exception-type="java.lang.NullPointerException"
                 channel="npeChannel"/>
</int:exception-type-router>

<int:channel id="illegalChannel" />
<int:channel id="npeChannel" />
```

8.1.4. Configuring a Generic Router

Spring Integration provides a generic router. You can use it for general-purpose routing (as opposed to the other routers provided by Spring Integration, each of which has some form of specialization).

Configuring a Content-based Router with XML

The `router` element provides a way to connect a router to an input channel and also accepts the optional `default-output-channel` attribute. The `ref` attribute references the bean name of a custom router implementation (which must extend `AbstractMessageRouter`). The following example shows three generic routers:

```
<int:router ref="payloadTypeRouter" input-channel="input1"
            default-output-channel="defaultOutput1"/>

<int:router ref="recipientListRouter" input-channel="input2"
            default-output-channel="defaultOutput2"/>

<int:router ref="customRouter" input-channel="input3"
            default-output-channel="defaultOutput3"/>

<beans:bean id="customRouterBean" class="org.foo.MyCustomRouter"/>
```

Alternatively, `ref` may point to a POJO that contains the `@Router` annotation (shown later), or you can combine the `ref` with an explicit method name. Specifying a method applies the same behavior described in the `@Router` annotation section, later in this document. The following example defines a router that points to a POJO in its `ref` attribute:

```
<int:router input-channel="input" ref="somePojo" method="someMethod"/>
```

We generally recommend using a `ref` attribute if the custom router implementation is referenced in other `<router>` definitions. However if the custom router implementation should be scoped to a single definition of the `<router>`, you can provide an inner bean definition, as the following example shows:

```
<int:router method="someMethod" input-channel="input3"
            default-output-channel="defaultOutput3">
    <beans:bean class="org.foo.MyCustomRouter"/>
</int:router>
```

Using both the `ref` attribute and an inner handler definition in the same `<router>` configuration is not allowed. Doing so creates an ambiguous condition and throws an exception.

If the `ref` attribute references a bean that extends `AbstractMessageProducingHandler` (such as routers provided by the framework itself), the configuration is optimized to reference the router directly. In this case, each `ref` attribute must refer to a separate bean instance (or a `prototype`-scoped bean) or use the inner `<bean/>` configuration type. However, this optimization applies only if you do not provide any router-specific attributes in the router XML definition. If you inadvertently reference the same message handler from multiple beans, you get a configuration exception.

The following example shows the equivalent router configured in Java:

```
@Bean
@Router(inputChannel = "routingChannel")
public AbstractMessageRouter myCustomRouter() {
    return new AbstractMessageRouter() {

        @Override
        protected Collection<MessageChannel> determineTargetChannels(Message<?>
message) {
            return // determine channel(s) for message
        }

    };
}
```

The following example shows the equivalent router configured by using the Java DSL:

```
@Bean
public IntegrationFlow routerFlow() {
    return IntegrationFlows.from("routingChannel")
            .route(myCustomRouter())
            .get();
}

public AbstractMessageRouter myCustomRouter() {
    return new AbstractMessageRouter() {

        @Override
        protected Collection<MessageChannel> determineTargetChannels(Message<?>
message) {
            return // determine channel(s) for message
        }

    };
}
```

Alternately, you can route on data from the message payload, as the following example shows:

```
@Bean
public IntegrationFlow routerFlow() {
    return IntegrationFlows.from("routingChannel")
            .route(String.class, p -> p.contains("foo") ? "fooChannel" :
"barChannel")
            .get();
}
```

8.1.5. Routers and the Spring Expression Language (SpEL)

Sometimes, the routing logic may be simple, and writing a separate class for it and configuring it as a bean may seem like overkill. As of Spring Integration 2.0, we offer an alternative that lets you use SpEL to implement simple computations that previously required a custom POJO router.

For more information about the Spring Expression Language, see the relevant chapter in the Spring Framework Reference Guide:

Generally, a SpEL expression is evaluated and its result is mapped to a channel, as the following example shows:

```
<int:router input-channel="inChannel" expression="payload.paymentType">
    <int:mapping value="CASH" channel="cashPaymentChannel"/>
    <int:mapping value="CREDIT" channel="authorizePaymentChannel"/>
    <int:mapping value="DEBIT" channel="authorizePaymentChannel"/>
</int:router>
```

The following example shows the equivalent router configured in Java:

```
@Router(inputChannel = "routingChannel")
@Bean
public ExpressionEvaluatingRouter router() {
    ExpressionEvaluatingRouter router = new ExpressionEvaluatingRouter(
"payload.paymentType");
    router.setChannelMapping("CASH", "cashPaymentChannel");
    router.setChannelMapping("CREDIT", "authorizePaymentChannel");
    router.setChannelMapping("DEBIT", "authorizePaymentChannel");
    return router;
}
```

The following example shows the equivalent router configured in the Java DSL:

```
@Bean
public IntegrationFlow routerFlow() {
    return IntegrationFlows.from("routingChannel")
        .route("payload.paymentType", r -> r
            .channelMapping("CASH", "cashPaymentChannel")
            .channelMapping("CREDIT", "authorizePaymentChannel")
            .channelMapping("DEBIT", "authorizePaymentChannel"))
        .get();
}
```

To simplify things even more, the SpEL expression may evaluate to a channel name, as the following expression shows:

```
<int:router input-channel="inChannel" expression="payload + 'Channel'"/>
```

In the preceding configuration, the result channel is computed by the SpEL expression, which concatenates the value of the `payload` with the literal `String`, 'Channel'.

Another virtue of SpEL for configuring routers is that an expression can return a `Collection`, effectively making every `<router>` a recipient list router. Whenever the expression returns multiple channel values, the message is forwarded to each channel. The following example shows such an expression:

```
<int:router input-channel="inChannel" expression="headers.channels"/>
```

In the above configuration, if the message includes a header with a name of 'channels' and the value of that header is a `List` of channel names, the message is sent to each channel in the list. You may also find collection projection and collection selection expressions useful when you need to select multiple channels. For further information, see:

- Collection Projection
- Collection Selection

Configuring a Router with Annotations

When using `@Router` to annotate a method, the method may return either a `MessageChannel` or a `String` type. In the latter case, the endpoint resolves the channel name as it does for the default output channel. Additionally, the method may return either a single value or a collection. If a collection is returned, the reply message is sent to multiple channels. To summarize, the following method signatures are all valid:

```
@Router
public MessageChannel route(Message message) {...}

@Router
public List<MessageChannel> route(Message message) {...}

@Router
public String route(Foo payload) {...}

@Router
public List<String> route(Foo payload) {...}
```

In addition to payload-based routing, a message may be routed based on metadata available within the message header as either a property or an attribute. In this case, a method annotated with `@Router` may include a parameter annotated with `@Header`, which is mapped to a header value as the following example shows and documented in Annotation Support:

```
@Router
public List<String> route(@Header("orderStatus") OrderStatus status)
```

For routing of XML-based Messages, including XPath support, see XML Support - Dealing with XML Payloads.

See also Message Routers in the Java DSL chapter for more information about router configuration.

8.1.6. Dynamic Routers

Spring Integration provides quite a few different router configurations for common content-based routing use cases as well as the option of implementing custom routers as POJOs. For example, `PayloadTypeRouter` provides a simple way to configure a router that computes channels based on the payload type of the incoming message while `HeaderValueRouter` provides the same convenience in configuring a router that computes channels by evaluating the value of a particular message Header. There are also expression-based (SpEL) routers, in which the channel is determined based on evaluating an expression. All of these type of routers exhibit some dynamic characteristics.

However, these routers all require static configuration. Even in the case of expression-based routers, the expression itself is defined as part of the router configuration, which means that the same expression operating on the same value always results in the computation of the same channel. This is acceptable in most cases, since such routes are well defined and therefore predictable. But there are times when we need to change router configurations dynamically so that message flows may be routed to a different channel.

For example, you might want to bring down some part of your system for maintenance and temporarily re-reroute messages to a different message flow. As another example, you may want to

introduce more granularity to your message flow by adding another route to handle a more concrete type of `java.lang.Number` (in the case of `PayloadTypeRouter`).

Unfortunately, with static router configuration to accomplish either of those goals, you would have to bring down your entire application, change the configuration of the router (change routes), and bring the application back up. This is obviously not a solution anyone wants.

The dynamic router pattern describes the mechanisms by which you can change or configure routers dynamically without bringing down the system or individual routers.

Before we get into the specifics of how Spring Integration supports dynamic routing, we need to consider the typical flow of a router:

1. Compute a channel identifier, which is a value calculated by the router once it receives the message. Typically, it is a String or an instance of the actual `MessageChannel`.
2. Resolve the channel identifier to a channel name. We describe specifics of this process later in this section.
3. Resolve the channel name to the actual `MessageChannel`

There is not much that can be done with regard to dynamic routing if Step 1 results in the actual instance of the `MessageChannel`, because the `MessageChannel` is the final product of any router's job. However, if the first step results in a channel identifier that is not an instance of `MessageChannel`, you have quite a few possible ways to influence the process of deriving the `MessageChannel`. Consider the following example of a payload type router:

```
<int:payload-type-router input-channel="routingChannel">
    <int:mapping type="java.lang.String"  channel="channel1" />
    <int:mapping type="java.lang.Integer" channel="channel2" />
</int:payload-type-router>
```

Within the context of a payload type router, the three steps mentioned earlier would be realized as follows:

1. Compute a channel identifier that is the fully qualified name of the payload type (for example, `java.lang.String`).
2. Resolve the channel identifier to a channel name, where the result of the previous step is used to select the appropriate value from the payload type mapping defined in the `mapping` element.
3. Resolve the channel name to the actual instance of the `MessageChannel` as a reference to a bean within the application context (which is hopefully a `MessageChannel`) identified by the result of the previous step.

In other words, each step feeds the next step until the process completes.

Now consider an example of a header value router:

```
<int:header-value-router input-channel="inputChannel" header-name="testHeader">
    <int:mapping value="foo" channel="fooChannel" />
    <int:mapping value="bar" channel="barChannel" />
</int:header-value-router>
```

Now we can consider how the three steps work for a header value router:

1. Compute a channel identifier that is the value of the header identified by the `header-name` attribute.
2. Resolve the channel identifier a to channel name, where the result of the previous step is used to select the appropriate value from the general mapping defined in the `mapping` element.
3. Resolve the channel name to the actual instance of the `MessageChannel` as a reference to a bean within the application context (which is hopefully a `MessageChannel`) identified by the result of the previous step.

The preceding two configurations of two different router types look almost identical. However, if you look at the alternate configuration of the `HeaderValueRouter` we clearly see that there is no `mapping` sub element, as the following listing shows:

```
<int:header-value-router input-channel="inputChannel" header-name="testHeader">
```

However, the configuration is still perfectly valid. So the natural question is what about the mapping in the second step?

The second step is now optional. If `mapping` is not defined, then the channel identifier value computed in the first step is automatically treated as the `channel name`, which is now resolved to the actual `MessageChannel`, as in the third step. What it also means is that the second step is one of the key steps to providing dynamic characteristics to the routers, since it introduces a process that lets you change the way channel identifier resolves to the channel name, thus influencing the process of determining the final instance of the `MessageChannel` from the initial channel identifier.

For example, in the preceding configuration, assume that the `testHeader` value is 'kermit', which is now a channel identifier (the first step). Since there is no mapping in this router, resolving this channel identifier to a channel name (the second step) is impossible and this channel identifier is now treated as the channel name. However, what if there was a mapping but for a different value? The end result would still be the same, because, if a new value cannot be determined through the process of resolving the channel identifier to a channel name, the channel identifier becomes the channel name.

All that is left is for the third step to resolve the channel name ('kermit') to an actual instance of the `MessageChannel` identified by this name. That basically involves a bean lookup for the provided name. Now all messages that contain the header-value pair as `testHeader=kermit` are going to be routed to a `MessageChannel` whose bean name (its `id`) is 'kermit'.

But what if you want to route these messages to the 'simpson' channel? Obviously changing a static configuration works, but doing so also requires bringing your system down. However, if you had access to the channel identifier map, you could introduce a new mapping where the header-value pair is now `kermit=simpson`, thus letting the second step treat 'kermit' as a channel identifier while resolving it to 'simpson' as the channel name.

The same obviously applies for `PayloadTypeRouter`, where you can now remap or remove a particular payload type mapping. In fact, it applies to every other router, including expression-based routers, since their computed values now have a chance to go through the second step to be resolved to the actual `channel name`.

Any router that is a subclass of the `AbstractMappingMessageRouter` (which includes most framework-defined routers) is a dynamic router, because the `channelMapping` is defined at the `AbstractMappingMessageRouter` level. That map's setter method is exposed as a public method along with the 'setChannelMapping' and 'removeChannelMapping' methods. These let you change, add, and remove router mappings at runtime, as long as you have a reference to the router itself. It also means that you could expose these same configuration options through JMX (see JMX Support) or the Spring Integration control bus (see Control Bus) functionality.

Falling back to the channel key as the channel name is flexible and convenient. However, if you don't trust the message creator, a malicious actor (who has knowledge of the system) could create a message that is routed to an unexpected channel. For example, if the key is set to the channel name of the router's input channel, such a message would be routed back to the router, eventually resulting in a stack overflow error. You may therefore wish to disable this feature (set the `channelKeyFallback` property to `false`), and change the mappings instead if needed.

Manage Router Mappings using the Control Bus

One way to manage the router mappings is through the control bus pattern, which exposes a control channel to which you can send control messages to manage and monitor Spring Integration components, including routers.

For more information about the control bus, see Control Bus.

Typically, you would send a control message asking to invoke a particular operation on a particular managed component (such as a router). The following managed operations (methods) are specific to changing the router resolution process:

- `public void setChannelMapping(String key, String channelName)`: Lets you add a new or modify an existing mapping between `channel identifier` and `channel name`
- `public void removeChannelMapping(String key)`: Lets you remove a particular channel mapping, thus disconnecting the relationship between `channel identifier` and `channel name`

Note that these methods can be used for simple changes (such as updating a single route or adding or removing a route). However, if you want to remove one route and add another, the updates are not atomic. This means that the routing table may be in an indeterminate state between the updates. Starting with version 4.0, you can now use the control bus to update the entire routing

table atomically. The following methods let you do so:

- `public Map<String, String>getChannelMappings()`: Returns the current mappings.
- `public void replaceChannelMappings(Properties channelMappings)`: Updates the mappings. Note that the `channelMappings` parameter is a `Properties` object. This arrangement lets a control bus command use the built-in `StringToPropertiesConverter`, as the following example shows:

```
"@'router.handler'.replaceChannelMappings('foo=qux \n baz=bar')"
```

Note that each mapping is separated by a newline character (`\n`). For programmatic changes to the map, we recommend that you use the `setChannelMappings` method, due to type-safety concerns. `replaceChannelMappings` ignores keys or values that are not `String` objects.

Manage Router Mappings by Using JMX

You can also use Spring's JMX support to expose a router instance and then use your favorite JMX client (for example, JConsole) to manage those operations (methods) for changing the router's configuration.

For more information about Spring Integration's JMX support, see JMX Support.

Routing Slip

Starting with version 4.1, Spring Integration provides an implementation of the routing slip enterprise integration pattern. It is implemented as a `routingSlip` message header, which is used to determine the next channel in `AbstractMessageProducingHandler` instances, when an `outputChannel` is not specified for the endpoint. This pattern is useful in complex, dynamic cases, when it can become difficult to configure multiple routers to determine message flow. When a message arrives at an endpoint that has no `output-channel`, the `routingSlip` is consulted to determine the next channel to which the message is sent. When the routing slip is exhausted, normal `replyChannel` processing resumes.

Configuration for the routing slip is presented as a `HeaderEnricher` option — a semicolon-separated routing slip that contains `path` entries, as the following example shows:

```
<util:properties id="properties">
    <beans:prop key="myRoutePath1">channel1</beans:prop>
    <beans:prop key="myRoutePath2">
request.headers[myRoutingSlipChannel]</beans:prop>
</util:properties>

<context:property-placeholder properties-ref="properties"/>

<header-enricher input-channel="input" output-channel="process">
    <routing-slip
        value="${myRoutePath1}; @routingSlipRoutingPojo.get(request, reply);
               routingSlipRoutingStrategy; ${myRoutePath2}; finishChannel"/>
</header-enricher>
```

The preceding example has:

- A `<context:property-placeholder>` configuration to demonstrate that the entries in the routing slip `path` can be specified as resolvable keys.
- The `<header-enricher>` `<routing-slip>` sub-element is used to populate the `RoutingSlipHeaderValueMessageProcessor` to the `HeaderEnricher` handler.
- The `RoutingSlipHeaderValueMessageProcessor` accepts a `String` array of resolved routing slip `path` entries and returns (from `processMessage()`) a `singletonMap` with the `path` as `key` and `0` as initial `routingSlipIndex`.

Routing Slip `path` entries can contain `MessageChannel` bean names, `RoutingSlipRouteStrategy` bean names, and Spring expressions (SpEL). The `RoutingSlipHeaderValueMessageProcessor` checks each routing slip `path` entry against the `BeanFactory` on the first `processMessage` invocation. It converts entries (which are not bean names in the application context) to `ExpressionEvaluatingRoutingSlipRouteStrategy` instances. `RoutingSlipRouteStrategy` entries are invoked multiple times, until they return null or an empty `String`.

Since the routing slip is involved in the `getOutputChannel` process, we have a request-reply context. The `RoutingSlipRouteStrategy` has been introduced to determine the next `outputChannel` that uses the `requestMessage` and the `reply` object. An implementation of this strategy should be registered as a bean in the application context, and its bean name is used in the routing slip `path`. The `ExpressionEvaluatingRoutingSlipRouteStrategy` implementation is provided. It accepts a SpEL expression and an internal `ExpressionEvaluatingRoutingSlipRouteStrategy.RequestAndReply` object is used as the root object of the evaluation context. This is to avoid the overhead of `EvaluationContext` creation for each `ExpressionEvaluatingRoutingSlipRouteStrategy.getNextPath()` invocation. It is a simple Java bean with two properties: `Message<?> request` and `Object reply`. With this expression implementation, we can specify routing slip `path` entries by using SpEL (for example, `@routingSlipRoutingPojo.get(request, reply)` and `request.headers[myRoutingSlipChannel]`) and avoid defining a bean for the `RoutingSlipRouteStrategy`.

The `requestMessage` argument is always a `Message<?>`. Depending on context, the reply object may be a `Message<?>`, an `AbstractIntegrationMessageBuilder`, or an arbitrary application domain object (when, for example, it is returned by a POJO method invoked by a service activator). In the first two cases, the usual `Message` properties (`payload` and `headers`) are available when using SpEL (or a Java implementation). For an arbitrary domain object, these properties are not available. For this reason, be careful when you use routing slips in conjunction with POJO methods if the result is used to determine the next path.

If a routing slip is involved in a distributed environment, we recommend not using inline expressions for the Routing Slip `path`. This recommendation applies to distributed environments such as cross-JVM applications, using a `request-reply` through a message broker (such asAMQP Support or JMS Support), or using a persistent `MessageStore` (Message Store) in the integration flow. The framework uses `RoutingSlipHeaderValueMessageProcessor` to convert them to `ExpressionEvaluatingRoutingSlipRouteStrategy` objects, and they are used in the `routingSlip` message header. Since this class is not `Serializable` (it cannot be, because it depends on the `BeanFactory`), the entire `Message` becomes non-serializable and, in any distributed operation, we end up with a `NotSerializableException`. To overcome this limitation, register an `ExpressionEvaluatingRoutingSlipRouteStrategy` bean with the desired SpEL and use its bean name in the routing slip `path` configuration.

For Java configuration, you can add a `RoutingSlipHeaderValueMessageProcessor` instance to the `HeaderEnricher` bean definition, as the following example shows:

```
@Bean
@Transformer(inputChannel = "routingSlipHeaderChannel")
public HeaderEnricher headerEnricher() {
    return new HeaderEnricher(Collections.singletonMap
(IntegrationMessageHeaderAccessor.ROUTING_SLIP,
            new RoutingSlipHeaderValueMessageProcessor("myRoutePath1",

"@routingSlipRoutingPojo.get(request, reply)",
                                                    "
routingSlipRoutingStrategy",

"request.headers[myRoutingSlipChannel]",
                                                    "finishChannel")));
}
```

The routing slip algorithm works as follows when an endpoint produces a reply and no `outputChannel` has been defined:

- The `routingSlipIndex` is used to get a value from the routing slip `path` list.

- If the value from `routingSlipIndex` is `String`, it is used to get a bean from `BeanFactory`.
- If a returned bean is an instance of `MessageChannel`, it is used as the next `outputChannel` and the `routingSlipIndex` is incremented in the reply message header (the routing slip `path` entries remain unchanged).
- If a returned bean is an instance of `RoutingSlipRouteStrategy` and its `getNextPath` does not return an empty `String`, that result is used as a bean name for the next `outputChannel`. The `routingSlipIndex` remains unchanged.
- If `RoutingSlipRouteStrategy.getNextPath` returns an empty `String` or `null`, the `routingSlipIndex` is incremented and the `getOutputChannelFromRoutingSlip` is invoked recursively for the next Routing Slip `path` item.
- If the next routing slip `path` entry is not a `String`, it must be an instance of `RoutingSlipRouteStrategy`.
- When the `routingSlipIndex` exceeds the size of the routing slip `path` list, the algorithm moves to the default behavior for the standard `replyChannel` header.

8.1.7. Process Manager Enterprise Integration Pattern

Enterprise integration patterns include the process manager pattern. You can now easily implement this pattern by using custom process manager logic encapsulated in a `RoutingSlipRouteStrategy` within the routing slip. In addition to a bean name, the `RoutingSlipRouteStrategy` can return any `MessageChannel` object, and there is no requirement that this `MessageChannel` instance be a bean in the application context. This way, we can provide powerful dynamic routing logic when there is no way to predict which channel should be used. A `MessageChannel` can be created within the `RoutingSlipRouteStrategy` and returned. A `FixedSubscriberChannel` with an associated `MessageHandler` implementation is a good combination for such cases. For example, you can route to a Reactive Streams, as the following example shows:

```
@Bean
public PollableChannel resultsChannel() {
    return new QueueChannel();
}
@Bean
public RoutingSlipRouteStrategy routeStrategy() {
    return (requestMessage, reply) -> requestMessage.getPayload() instanceof
String
            ? new FixedSubscriberChannel(m ->
            Mono.just((String) m.getPayload())
                    .map(String::toUpperCase)
                    .subscribe(v -> messagingTemplate().convertAndSend
(resultsChannel(), v)))
            : new FixedSubscriberChannel(m ->
            Mono.just((Integer) m.getPayload())
                    .map(v -> v * 2)
                    .subscribe(v -> messagingTemplate().convertAndSend
(resultsChannel(), v)));
}
```

8.2. Filter

Message filters are used to decide whether a `Message` should be passed along or dropped based on some criteria, such as a message header value or message content itself. Therefore, a message filter is similar to a router, except that, for each message received from the filter's input channel, that same message may or may not be sent to the filter's output channel. Unlike the router, it makes no decision regarding which message channel to send the message to but decides only whether to send the message at all.

As we describe later in this section, the filter also supports a discard channel. In certain cases, it can play the role of a very simple router (or "switch"), based on a boolean condition.

In Spring Integration, you can configure a message filter as a message endpoint that delegates to an implementation of the `MessageSelector` interface. That interface is itself quite simple, as the following listing shows:

```
public interface MessageSelector {

    boolean accept(Message<?> message);

}
```

The `MessageFilter` constructor accepts a selector instance, as the following example shows:

```
MessageFilter filter = new MessageFilter(someSelector);
```

In combination with the namespace and SpEL, you can configure powerful filters with very little Java code.

8.2.1. Configuring a Filter with XML

You can use the `<filter>` element is used to create a message-selecting endpoint. In addition to `input-channel` and `output-channel` attributes, it requires a `ref` attribute. The `ref` can point to a `MessageSelector` implementation, as the following example shows:

```
<int:filter input-channel="input" ref="selector" output-channel="output"/>

<bean id="selector" class="example.MessageSelectorImpl"/>
```

Alternatively, you can add the `method` attribute. In that case, the `ref` attribute may refer to any object. The referenced method may expect either the `Message` type or the payload type of inbound messages. The method must return a boolean value. If the method returns 'true', the message is sent to the output channel. The following example shows how to configure a filter that uses the `method` attribute:

```
<int:filter input-channel="input" output-channel="output"
    ref="exampleObject" method="someBooleanReturningMethod"/>

<bean id="exampleObject" class="example.SomeObject"/>
```

If the selector or adapted POJO method returns `false`, a few settings control the handling of the rejected message. By default (if configured as in the preceding example), rejected messages are silently dropped. If rejection should instead result in an error condition, set the `throw-exception-on-rejection` attribute to `true`, as the following example shows:

```
<int:filter input-channel="input" ref="selector"
    output-channel="output" throw-exception-on-rejection="true"/>
```

If you want rejected messages to be routed to a specific channel, provide that reference as the `discard-channel`, as the following example shows:

```
<int:filter input-channel="input" ref="selector"
    output-channel="output" discard-channel="rejectedMessages"/>
```

See also Advising Filters.

Message filters are commonly used in conjunction with a publish-subscribe channel. Many filter endpoints may be subscribed to the same channel, and they decide whether or not to pass the message to the next endpoint, which could be any of the supported types (such as a service activator). This provides a reactive alternative to the more proactive approach of using a message router with a single point-to-point input channel and multiple output channels.

We recommend using a `ref` attribute if the custom filter implementation is referenced in other `<filter>` definitions. However, if the custom filter implementation is scoped to a single `<filter>` element, you should provide an inner bean definition, as the following example shows:

```
<int:filter method="someMethod" input-channel="inChannel" output-channel=
"outChannel">
  <beans:bean class="org.foo.MyCustomFilter"/>
</filter>
```

Using both the `ref` attribute and an inner handler definition in the same `<filter>` configuration is not allowed, as it creates an ambiguous condition and throws an exception.

If the `ref` attribute references a bean that extends `MessageFilter` (such as filters provided by the framework itself), the configuration is optimized by injecting the output channel into the filter bean directly. In this case, each `ref` must be to a separate bean instance (or a `prototype`-scoped bean) or use the inner `<bean/>` configuration type. However, this optimization applies only if you do not provide any filter-specific attributes in the filter XML definition. If you inadvertently reference the same message handler from multiple beans, you get a configuration exception.

With the introduction of SpEL support, Spring Integration added the `expression` attribute to the filter element. It can be used to avoid Java entirely for simple filters, as the following example shows:

```
<int:filter input-channel="input" expression="payload.equals('nonsense')"/>
```

The string passed as the value of the expression attribute is evaluated as a SpEL expression with the message available in the evaluation context. If you must include the result of an expression in the scope of the application context, you can use the `#{}` notation, as defined in the SpEL reference documentation, as the following example shows:

```
<int:filter input-channel="input"
            expression="payload.matches(#{filterPatterns.nonsensePattern})"/>
```

If the expression itself needs to be dynamic, you can use an 'expression' sub-element. That provides a level of indirection for resolving the expression by its key from an `ExpressionSource`. That is a strategy interface that you can implement directly, or you can rely upon a version available in Spring Integration that loads expressions from a “resource bundle” and can check for modifications after a given number of seconds. All of this is demonstrated in the following configuration example, where the expression could be reloaded within one minute if the underlying file had been modified:

```
<int:filter input-channel="input" output-channel="output">
    <int:expression key="filterPatterns.example" source="myExpressions"/>
</int:filter>

<beans:bean id="myExpressions" id="myExpressions"
    class="o.s.i.expression.ReloadableResourceBundleExpressionSource">
    <beans:property name="basename" value="config/integration/expressions"/>
    <beans:property name="cacheSeconds" value="60"/>
</beans:bean>
```

If the `ExpressionSource` bean is named `expressionSource`, you need not provide the` source` attribute on the `<expression>` element. However, in the preceding example, we show it for completeness.

The 'config/integration/expressions.properties' file (or any more-specific version with a locale extension to be resolved in the typical way that resource-bundles are loaded) can contain a key/value pair, as the following example shows:

```
filterPatterns.example=payload > 100
```

All of these examples that use `expression` as an attribute or sub-element can also be applied within transformer, router, splitter, service-activator, and header-enricher elements. The semantics and role of the given component type would affect the interpretation of the evaluation result, in the same way that the return value of a method-invocation would be interpreted. For example, an expression can return strings that are to be treated as message channel names by a router component. However, the underlying functionality of evaluating the expression against the message as the root object and resolving bean names if prefixed with '@' is consistent across all of the core EIP components within Spring Integration.

8.2.2. Configuring a Filter with Annotations

The following example shows how to configure a filter by using annotations:

```
public class PetFilter {
    ...
    @Filter  ①
    public boolean dogsOnly(String input) {
        ...
    }
}
```

① An annotation indicating that this method is to be used as a filter. It must be specified if this class is to be used as a filter.

All of the configuration options provided by the XML element are also available for the `@Filter` annotation.

The filter can be either referenced explicitly from XML or, if the `@MessageEndpoint` annotation is defined on the class, detected automatically through classpath scanning.

See also Advising Endpoints Using Annotations.

8.3. Splitter

The splitter is a component whose role is to partition a message into several parts and send the resulting messages to be processed independently. Very often, they are upstream producers in a pipeline that includes an aggregator.

8.3.1. Programming Model

The API for performing splitting consists of one base class, `AbstractMessageSplitter`. It is a `MessageHandler` implementation that encapsulates features common to splitters, such as filling in the appropriate message headers (`CORRELATION_ID`, `SEQUENCE_SIZE`, and `SEQUENCE_NUMBER`) on the messages that are produced. This filling enables tracking down the messages and the results of their processing (in a typical scenario, these headers get copied to the messages that are produced by the various transforming endpoints). The values can then be used, for example, by a composed message processor.

The following example shows an excerpt from `AbstractMessageSplitter`:

```
public abstract class AbstractMessageSplitter
    extends AbstractReplyProducingMessageConsumer {
    ...
    protected abstract Object splitMessage(Message<?> message);

}
```

To implement a specific splitter in an application, you can extend `AbstractMessageSplitter` and implement the `splitMessage` method, which contains logic for splitting the messages. The return value can be one of the following:

- A `Collection` or an array of messages or an `Iterable` (or `Iterator`) that iterates over messages. In this case, the messages are sent as messages (after the `CORRELATION_ID`, `SEQUENCE_SIZE` and `SEQUENCE_NUMBER` are populated). Using this approach gives you more control — for example, to populate custom message headers as part of the splitting process.
- A `Collection` or an array of non-message objects or an `Iterable` (or `Iterator`) that iterates over non-message objects. It works like the prior case, except that each collection element is used as a message payload. Using this approach lets you focus on the domain objects without having to consider the messaging system and produces code that is easier to test.
- a `Message` or non-message object (but not a collection or an array). It works like the previous cases, except that a single message is sent out.

In Spring Integration, any POJO can implement the splitting algorithm, provided that it defines a method that accepts a single argument and has a return value. In this case, the return value of the method is interpreted as described earlier. The input argument might either be a `Message` or a simple POJO. In the latter case, the splitter receives the payload of the incoming message. We recommend this approach, because it decouples the code from the Spring Integration API and is typically easier to test.

Iterators

Starting with version 4.1, the `AbstractMessageSplitter` supports the `Iterator` type for the `value` to split. Note, in the case of an `Iterator` (or `Iterable`), we don't have access to the number of underlying items and the `SEQUENCE_SIZE` header is set to `0`. This means that the default `SequenceSizeReleaseStrategy` of an `<aggregator>` won't work and the group for the `CORRELATION_ID` from the `splitter` won't be released; it will remain as `incomplete`. In this case you should use an appropriate custom `ReleaseStrategy` or rely on `send-partial-result-on-expiry` together with `group-timeout` or a `MessageGroupStoreReaper`.

Starting with version 5.0, the `AbstractMessageSplitter` provides `protected obtainSizeIfPossible()` methods to allow the determination of the size of the `Iterable` and `Iterator` objects if that is possible. For example `XPathMessageSplitter` can determine the size of the underlying `NodeList` object. And starting with version 5.0.9, this method also properly returns a size of the `com.fasterxml.jackson.core.TreeNode`.

An `Iterator` object is useful to avoid the need for building an entire collection in the memory before

splitting. For example, when underlying items are populated from some external system (e.g. DataBase or FTP `MGET`) using iterations or streams.

Stream and Flux

Starting with version 5.0, the `AbstractMessageSplitter` supports the Java `Stream` and Reactive Streams `Publisher` types for the `value` to split. In this case, the target `Iterator` is built on their iteration functionality.

In addition, if the splitter's output channel is an instance of a `ReactiveStreamsSubscribableChannel`, the `AbstractMessageSplitter` produces a `Flux` result instead of an `Iterator`, and the output channel is subscribed to this `Flux` for back-pressure-based splitting on downstream flow demand.

Starting with version 5.2, the splitter supports a `discardChannel` option for sending those request messages for which a split function has returned an empty container (collection, array, stream, `Flux` etc.). In this case there is just no item to iterate for sending to the `outputChannel`. The `null` splitting result remains as an end of flow indicator.

8.3.2. Configuring a Splitter with XML

A splitter can be configured through XML as follows:

```
<int:channel id="inputChannel"/>

<int:splitter id="splitter"           ①
  ref="splitterBean"                  ②
  method="split"                      ③
  input-channel="inputChannel"        ④
  output-channel="outputChannel"      ⑤
  discard-channel="discardChannel" /> ⑥

<int:channel id="outputChannel"/>

<beans:bean id="splitterBean" class="sample.PojoSplitter"/>
```

① The ID of the splitter is optional.

② A reference to a bean defined in the application context. The bean must implement the splitting logic, as described in the earlier section. Optional. If a reference to a bean is not provided, it is assumed that the payload of the message that arrived on the `input-channel` is an implementation of `java.util.Collection` and the default splitting logic is applied to the collection, incorporating each individual element into a message and sending it to the `output-channel`.

③ The method (defined on the bean) that implements the splitting logic. Optional.

④ The input channel of the splitter. Required.

⑤ The channel to which the splitter sends the results of splitting the incoming message. Optional (because incoming messages can specify a reply channel themselves).

⑥ The channel to which the request message is sent in case of empty splitting result. Optional (the will stop as in case of `null` result).

We recommend using a `ref` attribute if the custom splitter implementation can be referenced in other `<splitter>` definitions. However if the custom splitter handler implementation should be scoped to a single definition of the `<splitter>`, you can configure an inner bean definition, as the following example follows:

```
<int:splitter id="testSplitter" input-channel="inChannel" method="split"
              output-channel="outChannel">
  <beans:bean class="org.foo.TestSplitter"/>
</int:splitter>
```

Using both a `ref` attribute and an inner handler definition in the same `<int:splitter>` configuration is not allowed, as it creates an ambiguous condition and results in an exception being thrown.

If the `ref` attribute references a bean that extends `AbstractMessageProducingHandler` (such as splitters provided by the framework itself), the configuration is optimized by injecting the output channel into the handler directly. In this case, each `ref` must be a separate bean instance (or a `prototype`-scoped bean) or use the inner `<bean/>` configuration type. However, this optimization applies only if you do not provide any splitter-specific attributes in the splitter XML definition. If you inadvertently reference the same message handler from multiple beans, you get a configuration exception.

8.3.3. Configuring a Splitter with Annotations

The `@Splitter` annotation is applicable to methods that expect either the `Message` type or the message payload type, and the return values of the method should be a `Collection` of any type. If the returned values are not actual `Message` objects, each item is wrapped in a `Message` as the payload of the `Message`. Each resulting `Message` is sent to the designated output channel for the endpoint on which the `@Splitter` is defined.

The following example shows how to configure a splitter by using the `@Splitter` annotation:

```
@Splitter
List<LineItem> extractItems(Order order) {
    return order.getItems()
}
```

See also Advising Endpoints Using Annotations.

See also Splitters in the Java DSL chapter.

8.4. Aggregator

Basically a mirror-image of the splitter, the aggregator is a type of message handler that receives multiple messages and combines them into a single message. In fact, an aggregator is often a downstream consumer in a pipeline that includes a splitter.

Technically, the aggregator is more complex than a splitter, because it is stateful. It must hold the messages to be aggregated and determine when the complete group of messages is ready to be aggregated. In order to do so, it requires a `MessageStore`.

8.4.1. Functionality

The Aggregator combines a group of related messages, by correlating and storing them, until the group is deemed to be complete. At that point, the aggregator creates a single message by processing the whole group and sends the aggregated message as output.

Implementing an aggregator requires providing the logic to perform the aggregation (that is, the creation of a single message from many). Two related concepts are correlation and release.

Correlation determines how messages are grouped for aggregation. In Spring Integration, correlation is done by default, based on the `IntegrationMessageHeaderAccessor.CORRELATION_ID` message header. Messages with the same `IntegrationMessageHeaderAccessor.CORRELATION_ID` are grouped together. However, you can customize the correlation strategy to allow other ways of specifying how the messages should be grouped together. To do so, you can implement a `CorrelationStrategy` (covered later in this chapter).

To determine the point at which a group of messages is ready to be processed, a `ReleaseStrategy` is consulted. The default release strategy for the aggregator releases a group when all messages included in a sequence are present, based on the `IntegrationMessageHeaderAccessor.SEQUENCE_SIZE` header. You can override this default strategy by providing a reference to a custom `ReleaseStrategy` implementation.

8.4.2. Programming Model

The Aggregation API consists of a number of classes:

- The interface `MessageGroupProcessor`, and its subclasses: `MethodInvokingAggregatingMessageGroupProcessor` and `ExpressionEvaluatingMessageGroupProcessor`
- The `ReleaseStrategy` interface and its default implementation: `SimpleSequenceSizeReleaseStrategy`
- The `CorrelationStrategy` interface and its default implementation: `HeaderAttributeCorrelationStrategy`

`AggregatingMessageHandler`

The `AggregatingMessageHandler` (a subclass of `AbstractCorrelatingMessageHandler`) is a `MessageHandler` implementation, encapsulating the common functionality of an aggregator (and other correlating use cases), which are as follows:

- Correlating messages into a group to be aggregated
- Maintaining those messages in a `MessageStore` until the group can be released
- Deciding when the group can be released
- Aggregating the released group into a single message
- Recognizing and responding to an expired group

The responsibility for deciding how the messages should be grouped together is delegated to a `CorrelationStrategy` instance. The responsibility for deciding whether the message group can be released is delegated to a `ReleaseStrategy` instance.

The following listing shows a brief highlight of the base `AbstractAggregatingMessageGroupProcessor` (the responsibility for implementing the `aggregatePayloads` method is left to the developer):

```
public abstract class AbstractAggregatingMessageGroupProcessor
              implements MessageGroupProcessor {

    protected Map<String, Object> aggregateHeaders(MessageGroup group) {
        // default implementation exists
    }

    protected abstract Object aggregatePayloads(MessageGroup group, Map<String,
Object> defaultHeaders);

}
```

See `DefaultAggregatingMessageGroupProcessor`, `ExpressionEvaluatingMessageGroupProcessor` and `MethodInvokingMessageGroupProcessor` as out-of-the-box implementations of the `AbstractAggregatingMessageGroupProcessor`.

Starting with version 5.2, a `Function<MessageGroup, Map<String, Object>>` strategy is available for the `AbstractAggregatingMessageGroupProcessor` to merge and compute (aggregate) headers for an output message. The `DefaultAggregateHeadersFunction` implementation is available with logic that returns all headers that have no conflicts among the group; an absent header on one or more messages within the group is not considered a conflict. Conflicting headers are omitted. Along with the newly introduced `DelegatingMessageGroupProcessor`, this function is used for any arbitrary (non-`AbstractAggregatingMessageGroupProcessor`) `MessageGroupProcessor` implementation. Essentially, the framework injects a provided function into an `AbstractAggregatingMessageGroupProcessor` instance and wraps all other implementations into a `DelegatingMessageGroupProcessor`. The difference in logic between the `AbstractAggregatingMessageGroupProcessor` and the `DelegatingMessageGroupProcessor` that the latter doesn't compute headers in advance, before calling the delegate strategy, and doesn't invoke the function if the delegate returns a `Message` or `AbstractIntegrationMessageBuilder`. In that case, the framework assumes that the target implementation has taken care of producing a proper set of headers populated into the returned result. The `Function<MessageGroup, Map<String, Object>>` strategy is available as the `headers-function` reference attribute for XML configuration, as the `AggregatorSpec.headersFunction()` option for the Java DSL and as `AggregatorFactoryBean.setHeadersFunction()` for plain Java configuration.

The `CorrelationStrategy` is owned by the `AbstractCorrelatingMessageHandler` and has a default value based on the `IntegrationMessageHeaderAccessor.CORRELATION_ID` message header, as the following example shows:

```
public AbstractCorrelatingMessageHandler(MessageGroupProcessor processor,
MessageGroupStore store,
        CorrelationStrategy correlationStrategy, ReleaseStrategy releaseStrategy)
{
    ...
    this.correlationStrategy = correlationStrategy == null ?
        new HeaderAttributeCorrelationStrategy(IntegrationMessageHeaderAccessor
.CORRELATION_ID) : correlationStrategy;
    this.releaseStrategy = releaseStrategy == null ? new
SimpleSequenceSizeReleaseStrategy() : releaseStrategy;
    ...
}
```

As for the actual processing of the message group, the default implementation is the `DefaultAggregatingMessageGroupProcessor`. It creates a single `Message` whose payload is a `List` of the payloads received for a given group. This works well for simple scatter-gather implementations with a splitter, a publish-subscribe channel, or a recipient list router upstream.

When using a publish-subscribe channel or a recipient list router in this type of scenario, be sure to enable the `apply-sequence` flag. Doing so adds the necessary headers: `CORRELATION_ID`, `SEQUENCE_NUMBER`, and `SEQUENCE_SIZE`. That behavior is enabled by default for splitters in Spring Integration, but it is not enabled for publish-subscribe channels or for recipient list routers because those components may be used in a variety of contexts in which these headers are not necessary.

When implementing a specific aggregator strategy for an application, you can extend `AbstractAggregatingMessageGroupProcessor` and implement the `aggregatePayloads` method. However, there are better solutions, less coupled to the API, for implementing the aggregation logic, which can be configured either through XML or through annotations.

In general, any POJO can implement the aggregation algorithm if it provides a method that accepts a single `java.util.List` as an argument (parameterized lists are supported as well). This method is invoked for aggregating messages as follows:

- If the argument is a `java.util.Collection<T>` and the parameter type T is assignable to `Message`, the whole list of messages accumulated for aggregation is sent to the aggregator.
- If the argument is a non-parameterized `java.util.Collection` or the parameter type is not assignable to `Message`, the method receives the payloads of the accumulated messages.
- If the return type is not assignable to `Message`, it is treated as the payload for a `Message` that is automatically created by the framework.

In the interest of code simplicity and promoting best practices such as low coupling, testability, and others, the preferred way of implementing the aggregation logic is through a POJO and using the XML or annotation support for configuring it in the application.

Starting with version 5.3, after processing message group, an `AbstractCorrelatingMessageHandler` performs a `MessageBuilder.popSequenceDetails()` message headers modification for the proper splitter-aggregator scenario with several nested levels. It is done only if the message group release result is not a collection of messages. In that case a target `MessageGroupProcessor` is responsible for the `MessageBuilder.popSequenceDetails()` call while building those messages.

If the `MessageGroupProcessor` returns a `Message`, a `MessageBuilder.popSequenceDetails()` will be performed on the output message only if the `sequenceDetails` matches with first message in group. (Previously this has been done only if a plain payload or an `AbstractIntegrationMessageBuilder` has been returned from the `MessageGroupProcessor`.)

This functionality can be controlled by a new `popSequence` `boolean` property, so the `MessageBuilder.popSequenceDetails()` can be disabled in some scenarios when correlation details have not been populated by the standard splitter. This property, essentially, undoes what has been done by the nearest upstream `applySequence = true` in the `AbstractMessageSplitter`. See Splitter for more information.

The `SimpleMessageGroup.getMessages()` method returns an `unmodifiableCollection`. Therefore, if your aggregating POJO method has a `Collection<Message>` parameter, the argument passed in is exactly that `Collection` instance and, when you use a `SimpleMessageStore` for the aggregator, that original `Collection<Message>` is cleared after releasing the group. Consequently, the `Collection<Message>` variable in the POJO is cleared too, if it is passed out of the aggregator. If you wish to simply release that collection as-is for further processing, you must build a new `Collection` (for example, `new ArrayList<Message>(messages)`). Starting with version 4.3, the framework no longer copies the messages to a new collection, to avoid undesired extra object creation.

If the `processMessageGroup` method of the `MessageGroupProcessor` returns a collection, it must be a collection of `Message<?>` objects. In this case, the messages are individually released. Prior to version 4.2, it was not possible to provide a `MessageGroupProcessor` by using XML configuration. Only POJO methods could be used for aggregation. Now, if the framework detects that the referenced (or inner) bean implements `MessageProcessor`, it is used as the aggregator's output processor.

If you wish to release a collection of objects from a custom `MessageGroupProcessor` as the payload of a message, your class should extend `AbstractAggregatingMessageGroupProcessor` and implement `aggregatePayloads()`.

Also, since version 4.2, a `SimpleMessageGroupProcessor` is provided. It returns the collection of messages from the group, which, as indicated earlier, causes the released messages to be sent individually.

This lets the aggregator work as a message barrier, where arriving messages are held until the release strategy fires and the group is released as a sequence of individual messages.

`ReleaseStrategy`

The `ReleaseStrategy` interface is defined as follows:

```
public interface ReleaseStrategy {

  boolean canRelease(MessageGroup group);

}
```

In general, any POJO can implement the completion decision logic if it provides a method that accepts a single `java.util.List` as an argument (parameterized lists are supported as well) and returns a boolean value. This method is invoked after the arrival of each new message, to decide whether the group is complete or not, as follows:

- If the argument is a `java.util.List<T>` and the parameter type `T` is assignable to `Message`, the whole list of messages accumulated in the group is sent to the method.
- If the argument is a non-parametrized `java.util.List` or the parameter type is not assignable to `Message`, the method receives the payloads of the accumulated messages.
- The method must return `true` if the message group is ready for aggregation or false otherwise.

The following example shows how to use the `@ReleaseStrategy` annotation for a `List` of type `Message`:

```
public class MyReleaseStrategy {

    @ReleaseStrategy
    public boolean canMessagesBeReleased(List<Message<?>>) {...}
}
```

The following example shows how to use the `@ReleaseStrategy` annotation for a `List` of type `String`:

```
public class MyReleaseStrategy {

    @ReleaseStrategy
    public boolean canMessagesBeReleased(List<String>) {...}
}
```

Based on the signatures in the preceding two examples, the POJO-based release strategy is passed a `Collection` of not-yet-released messages (if you need access to the whole `Message`) or a `Collection` of payload objects (if the type parameter is anything other than `Message`). This satisfies the majority of use cases. However if, for some reason, you need to access the full `MessageGroup`, you should provide an implementation of the `ReleaseStrategy` interface.

When handling potentially large groups, you should understand how these methods are invoked, because the release strategy may be invoked multiple times before the group is released. The most efficient is an implementation of `ReleaseStrategy`, because the aggregator can invoke it directly. The second most efficient is a POJO method with a `Collection<Message<?>>` parameter type. The least efficient is a POJO method with a `Collection<Something>` type. The framework has to copy the payloads from the messages in the group into a new collection (and possibly attempt conversion on the payloads to `Something`) every time the release strategy is called. Using `Collection<?>` avoids the conversion but still requires creating the new `Collection`.

For these reasons, for large groups, we recommended that you implement `ReleaseStrategy`.

When the group is released for aggregation, all its not-yet-released messages are processed and removed from the group. If the group is also complete (that is, if all messages from a sequence have arrived or if there is no sequence defined), then the group is marked as complete. Any new messages for this group are sent to the discard channel (if defined). Setting `expire-groups-upon-completion` to `true` (the default is `false`) removes the entire group, and any new messages (with the same correlation ID as the removed group) form a new group. You can release partial sequences by using a `MessageGroupStoreReaper` together with `send-partial-result-on-expiry` being set to `true`.

To facilitate discarding of late-arriving messages, the aggregator must maintain state about the group after it has been released. This can eventually cause out-of-memory conditions. To avoid such situations, you should consider configuring a `MessageGroupStoreReaper` to remove the group metadata. The expiry parameters should be set to expire groups once a point has been reach after after which late messages are not expected to arrive. For information about configuring a reaper, see Managing State in an Aggregator: `MessageGroupStore`.

Spring Integration provides an implementation for `ReleaseStrategy`: `SimpleSequenceSizeReleaseStrategy`. This implementation consults the `SEQUENCE_NUMBER` and `SEQUENCE_SIZE` headers of each arriving message to decide when a message group is complete and ready to be aggregated. As shown earlier, it is also the default strategy.

Before version 5.0, the default release strategy was `SequenceSizeReleaseStrategy`, which does not perform well with large groups. With that strategy, duplicate sequence numbers are detected and rejected. This operation can be expensive.

If you are aggregating large groups, you don't need to release partial groups, and you don't need to detect/reject duplicate sequences, consider using the `SimpleSequenceSizeReleaseStrategy` instead - it is much more efficient for these use cases, and is the default since *version 5.0* when partial group release is not specified.

Aggregating Large Groups

The 4.3 release changed the default `Collection` for messages in a `SimpleMessageGroup` to `HashSet` (it was previously a `BlockingQueue`). This was expensive when removing individual messages from

large groups (an O(n) linear scan was required). Although the hash set is generally much faster to remove, it can be expensive for large messages, because the hash has to be calculated on both inserts and removes. If you have messages that are expensive to hash, consider using some other collection type. As discussed in Using `MessageGroupFactory`, a `SimpleMessageGroupFactory` is provided so that you can select the `Collection` that best suits your needs. You can also provide your own factory implementation to create some other `Collection<Message<?>>`.

The following example shows how to configure an aggregator with the previous implementation and a `SimpleSequenceSizeReleaseStrategy`:

```
<int:aggregator input-channel="aggregate"
    output-channel="out" message-store="store" release-strategy="releaser" />

<bean id="store" class="org.springframework.integration.store.SimpleMessageStore">
    <property name="messageGroupFactory">
        <bean class=
"org.springframework.integration.store.SimpleMessageGroupFactory">
            <constructor-arg value="BLOCKING_QUEUE"/>
        </bean>
    </property>
</bean>

<bean id="releaser" class="SimpleSequenceSizeReleaseStrategy" />
```

Correlation Strategy

The `CorrelationStrategy` interface is defined as follows:

```
public interface CorrelationStrategy {

  Object getCorrelationKey(Message<?> message);

}
```

The method returns an `Object` that represents the correlation key used for associating the message with a message group. The key must satisfy the criteria used for a key in a `Map` with respect to the implementation of `equals()` and `hashCode()`.

In general, any POJO can implement the correlation logic, and the rules for mapping a message to a method's argument (or arguments) are the same as for a `ServiceActivator` (including support for `@Header` annotations). The method must return a value, and the value must not be `null`.

Spring Integration provides an implementation for `CorrelationStrategy`: `HeaderAttributeCorrelationStrategy`. This implementation returns the value of one of the message headers (whose name is specified by a constructor argument) as the correlation key. By default, the

correlation strategy is a `HeaderAttributeCorrelationStrategy` that returns the value of the `CORRELATION_ID` header attribute. If you have a custom header name you would like to use for correlation, you can configure it on an instance of `HeaderAttributeCorrelationStrategy` and provide that as a reference for the aggregator's correlation strategy.

Lock Registry

Changes to groups are thread safe. So, when you send messages for the same correlation ID concurrently, only one of them will be processed in the aggregator, making it effectively as a **single-threaded per message group**. A `LockRegistry` is used to obtain a lock for the resolved correlation ID. A `DefaultLockRegistry` is used by default (in-memory). For synchronizing updates across servers where a shared `MessageGroupStore` is being used, you must configure a shared lock registry.

Avoiding Deadlocks

As discussed above, when message groups are mutated (messages added or released) a lock is held.

Consider the following flow:

```
...->aggregator1-> ... ->aggregator2-> ...
```

If there are multiple threads, **and the aggregators share a common lock registry**, it is possible to get a deadlock. This will cause hung threads and `jstack <pid>` might present a result such as:

```
Found one Java-level deadlock:
=============================
"t2":
  waiting for ownable synchronizer 0x000000076c1cbfa0, (a
java.util.concurrent.locks.ReentrantLock$NonfairSync),
  which is held by "t1"
"t1":
  waiting for ownable synchronizer 0x000000076c1ccc00, (a
java.util.concurrent.locks.ReentrantLock$NonfairSync),
  which is held by "t2"
```

There are several ways to avoid this problem:

- ensure each aggregator has its own lock registry (this can be a shared registry across application instances but two or more aggregators in the flow must each have a distinct registry)
- use an `ExecutorChannel` or `QueueChannel` as the output channel of the aggregator so that the downstream flow runs on a new thread
- starting with version 5.1.1, set the `releaseLockBeforeSend` aggregator property to `true`

This problem can also be caused if, for some reason, the output of a single aggregator is eventually routed back to the same aggregator. Of course, the first solution above does not apply in this case.

8.4.3. Configuring an Aggregator in Java DSL

See Aggregators and Resequencers for how to configure an aggregator in Java DSL.

Configuring an Aggregator with XML

Spring Integration supports the configuration of an aggregator with XML through the `<aggregator/>` element. The following example shows an example of an aggregator:

```
<channel id="inputChannel"/>

<int:aggregator id="myAggregator"                                ①
        auto-startup="true"                                      ②
        input-channel="inputChannel"                             ③
        output-channel="outputChannel"                           ④
        discard-channel="throwAwayChannel"                       ⑤
        message-store="persistentMessageStore"                   ⑥
        order="1"                                                ⑦
        send-partial-result-on-expiry="false"                    ⑧
        send-timeout="1000"                                      ⑨

        correlation-strategy="correlationStrategyBean"           ⑩
        correlation-strategy-method="correlate"                  ⑪
        correlation-strategy-expression="headers['foo']"         ⑫

        ref="aggregatorBean"                                     ⑬
        method="aggregate"                                       ⑭

        release-strategy="releaseStrategyBean"                   ⑮
        release-strategy-method="release"                        ⑯
        release-strategy-expression="size() == 5"                ⑰

        expire-groups-upon-completion="false"                    ⑱
        empty-group-min-timeout="60000"                          ⑲

        lock-registry="lockRegistry"                             ⑳

        group-timeout="60000"
        group-timeout-expression="size() ge 2 ? 100 : -1"
        expire-groups-upon-timeout="true"

        scheduler="taskScheduler" >
            <expire-transactional/>
            <expire-advice-chain/>
</aggregator>

<int:channel id="outputChannel"/>

<int:channel id="throwAwayChannel"/>

<bean id="persistentMessageStore" class=
"org.springframework.integration.jdbc.store.JdbcMessageStore">
    <constructor-arg ref="dataSource"/>
</bean>

<bean id="aggregatorBean" class="sample.PojoAggregator"/>

<bean id="releaseStrategyBean" class="sample.PojoReleaseStrategy"/>
```

```
<bean id="correlationStrategyBean" class="sample.PojoCorrelationStrategy"/>
```

① The id of the aggregator is optional.

② Lifecycle attribute signaling whether the aggregator should be started during application context startup. Optional (the default is 'true').

③ The channel from which where aggregator receives messages. Required.

④ The channel to which the aggregator sends the aggregation results. Optional (because incoming messages can themselves specify a reply channel in the 'replyChannel' message header).

⑤ The channel to which the aggregator sends the messages that timed out (if `send-partial-result-on-expiry` is `false`). Optional.

⑥ A reference to a `MessageGroupStore` used to store groups of messages under their correlation key until they are complete. Optional. By default, it is a volatile in-memory store. See Message Store for more information.

⑦ The order of this aggregator when more than one handle is subscribed to the same `DirectChannel` (use for load-balancing purposes). Optional.

⑧ Indicates that expired messages should be aggregated and sent to the 'output-channel' or 'replyChannel' once their containing `MessageGroup` is expired (see `MessageGroupStore.expireMessageGroups(long)`). One way of expiring a `MessageGroup` is by configuring a `MessageGroupStoreReaper`. However you can alternatively expire `MessageGroup` by calling `MessageGroupStore.expireMessageGroups(timeout)`. You can accomplish that through a Control Bus operation or, if you have a reference to the `MessageGroupStore` instance, by invoking `expireMessageGroups(timeout)`. Otherwise, by itself, this attribute does nothing. It serves only as an indicator of whether to discard or send to the output or reply channel any messages that are still in the `MessageGroup` that is about to be expired. Optional (the default is `false`). NOTE: This attribute might more properly be called `send-partial-result-on-timeout`, because the group may not actually expire if `expire-groups-upon-timeout` is set to `false`.

⑨ The timeout interval to wait when sending a reply `Message` to the `output-channel` or `discard-channel`. Defaults to `-1`, which results in blocking indefinitely. It is applied only if the output channel has some 'sending' limitations, such as a `QueueChannel` with a fixed 'capacity'. In this case, a `MessageDeliveryException` is thrown. For `AbstractSubscribableChannel` implementations, the `send-timeout` is ignored . For `group-timeout(-expression)`, the `MessageDeliveryException` from the scheduled expire task leads this task to be rescheduled. Optional.

⑩ A reference to a bean that implements the message correlation (grouping) algorithm. The bean can be an implementation of the `CorrelationStrategy` interface or a POJO. In the latter case, the `correlation-strategy-method` attribute must be defined as well. Optional (by default, the aggregator uses the `IntegrationMessageHeaderAccessor.CORRELATION_ID` header).

⑪ A method defined on the bean referenced by `correlation-strategy`. It implements the correlation decision algorithm. Optional, with restrictions (`correlation-strategy` must be present).

⑫ A SpEL expression representing the correlation strategy. Example: `"headers['something']"`.

Only one of `correlation-strategy` or `correlation-strategy-expression` is allowed.

⑬ A reference to a bean defined in the application context. The bean must implement the aggregation logic, as described earlier. Optional (by default, the list of aggregated messages becomes a payload of the output message).

⑭ A method defined on the bean referenced by the `ref` attribute. It implements the message aggregation algorithm. Optional (it depends on `ref` attribute being defined).

⑮ A reference to a bean that implements the release strategy. The bean can be an implementation of the `ReleaseStrategy` interface or a POJO. In the latter case, the `release-strategy-method` attribute must be defined as well. Optional (by default, the aggregator uses the `IntegrationMessageHeaderAccessor.SEQUENCE_SIZE` header attribute).

⑯ A method defined on the bean referenced by the `release-strategy` attribute. It implements the completion decision algorithm. Optional, with restrictions (`release-strategy` must be present).

⑰ A SpEL expression representing the release strategy. The root object for the expression is a `MessageGroup`. Example: `"size() == 5"`. Only one of `release-strategy` or `release-strategy-expression` is allowed.

⑱ When set to `true` (the default is `false`), completed groups are removed from the message store, letting subsequent messages with the same correlation form a new group. The default behavior is to send messages with the same correlation as a completed group to the `discard-channel`.

⑲ Applies only if a `MessageGroupStoreReaper` is configured for the `MessageStore` of the `<aggregator>`. By default, when a `MessageGroupStoreReaper` is configured to expire partial groups, empty groups are also removed. Empty groups exist after a group is normally released. The empty groups enable the detection and discarding of late-arriving messages. If you wish to expire empty groups on a longer schedule than expiring partial groups, set this property. Empty groups are then not removed from the `MessageStore` until they have not been modified for at least this number of milliseconds. Note that the actual time to expire an empty group is also affected by the reaper's `timeout` property, and it could be as much as this value plus the timeout.

⑳ A reference to a `org.springframework.integration.util.LockRegistry` bean. It used to obtain a `Lock` based on the `groupId` for concurrent operations on the `MessageGroup`. By default, an internal `DefaultLockRegistry` is used. Use of a distributed `LockRegistry`, such as the `ZookeeperLockRegistry`, ensures only one instance of the aggregator can operate on a group concurrently. See Redis Lock Registry, Gemfire Lock Registry, and Zookeeper Lock Registry for more information.

A timeout (in milliseconds) to force the `MessageGroup` complete when the `ReleaseStrategy` does not release the group when the current message arrives. This attribute provides a built-in time-based release strategy for the aggregator when there is a need to emit a partial result (or discard the group) if a new message does not arrive for the `MessageGroup` within the timeout which counts from the time the last message arrived. To set up a timeout which counts from the time the `MessageGroup` was created see `group-timeout-expression` information. When a new message arrives at the aggregator, any existing `ScheduledFuture<?>` for its `MessageGroup` is canceled. If the `ReleaseStrategy` returns `false` (meaning do not release) and `groupTimeout > 0`, a new task is scheduled to expire the group. We do not advise setting this attribute to zero (or

a negative value). Doing so effectively disables the aggregator, because every message group is immediately completed. You can, however, conditionally set it to zero (or a negative value) by using an expression. See `group-timeout-expression` for information. The action taken during the completion depends on the `ReleaseStrategy` and the `send-partial-group-on-expiry` attribute. See Aggregator and Group Timeout for more information. It is mutually exclusive with 'group-timeout-expression' attribute.

The SpEL expression that evaluates to a `groupTimeout` with the `MessageGroup` as the `#root` evaluation context object. Used for scheduling the `MessageGroup` to be forced complete. If the expression evaluates to `null`, the completion is not scheduled. If it evaluates to zero, the group is completed immediately on the current thread. In effect, this provides a dynamic `group-timeout` property. As an example, if you wish to forcibly complete a `MessageGroup` after 10 seconds have elapsed since the time the group was created you might consider using the following SpEL expression: `timestamp + 10000 - T(System).currentTimeMillis()` where `timestamp` is provided by `MessageGroup.getTimestamp()` as the `MessageGroup` here is the `#root` evaluation context object. Bear in mind however that the group creation time might differ from the time of the first arrived message depending on other group expiration properties' configuration. See `group-timeout` for more information. Mutually exclusive with 'group-timeout' attribute.

When a group is completed due to a timeout (or by a `MessageGroupStoreReaper`), the group is expired (completely removed) by default. Late arriving messages start a new group. You can set this to `false` to complete the group but have its metadata remain so that late arriving messages are discarded. Empty groups can be expired later using a `MessageGroupStoreReaper` together with the `empty-group-min-timeout` attribute. It defaults to 'true'.

A `TaskScheduler` bean reference to schedule the `MessageGroup` to be forced complete if no new message arrives for the `MessageGroup` within the `groupTimeout`. If not provided, the default scheduler (`taskScheduler`) registered in the `ApplicationContext` (`ThreadPoolTaskScheduler`) is used. This attribute does not apply if `group-timeout` or `group-timeout-expression` is not specified.

Since version 4.1. It lets a transaction be started for the `forceComplete` operation. It is initiated from a `group-timeout(-expression)` or by a `MessageGroupStoreReaper` and is not applied to the normal `add`, `release`, and `discard` operations. Only this sub-element or `<expire-advice-chain/>` is allowed.

Since *version 4.1*. It allows the configuration of any `Advice` for the `forceComplete` operation. It is initiated from a `group-timeout(-expression)` or by a `MessageGroupStoreReaper` and is not applied to the normal `add`, `release`, and `discard` operations. Only this sub-element or `<expire-transactional/>` is allowed. A transaction `Advice` can also be configured here by using the Spring `tx` namespace.

Expiring Groups

There are two attributes related to expiring (completely removing) groups. When a group is expired, there is no record of it, and, if a new message arrives with the same correlation, a new group is started. When a group is completed (without expiry), the empty group remains and late-arriving messages are discarded. Empty groups can be removed later by using a `MessageGroupStoreReaper` in combination with the `empty-group-min-timeout` attribute.

`expire-groups-upon-completion` relates to "normal" completion when the `ReleaseStrategy` releases the group. This defaults to `false`.

If a group is not completed normally but is released or discarded because of a timeout, the group is normally expired. Since version 4.1, you can control this behavior by using `expire-groups-upon-timeout`. It defaults to `true` for backwards compatibility.

When a group is timed out, the `ReleaseStrategy` is given one more opportunity to release the group. If it does so and `expire-groups-upon-timeout` is false, expiration is controlled by `expire-groups-upon-completion`. If the group is not released by the release strategy during timeout, then the expiration is controlled by the `expire-groups-upon-timeout`. Timed-out groups are either discarded or a partial release occurs (based on `send-partial-result-on-expiry`).

Since version 5.0, empty groups are also scheduled for removal after `empty-group-min-timeout`. If `expireGroupsUponCompletion == false` and `minimumTimeoutForEmptyGroups > 0`, the task to remove the group is scheduled when normal or partial sequences release happens.

We generally recommend using a `ref` attribute if a custom aggregator handler implementation may be referenced in other `<aggregator>` definitions. However, if a custom aggregator implementation is only being used by a single definition of the `<aggregator>`, you can use an inner bean definition (starting with version 1.0.3) to configure the aggregation POJO within the `<aggregator>` element, as the following example shows:

```
<aggregator input-channel="input" method="sum" output-channel="output">
    <beans:bean class="org.foo.PojoAggregator"/>
</aggregator>
```

Using both a `ref` attribute and an inner bean definition in the same `<aggregator>` configuration is not allowed, as it creates an ambiguous condition. In such cases, an Exception is thrown.

The following example shows an implementation of the aggregator bean:

```
public class PojoAggregator {

  public Long add(List<Long> results) {
    long total = 0l;
    for (long partialResult: results) {
      total += partialResult;
    }
    return total;
  }
}
```

An implementation of the completion strategy bean for the preceding example might be as follows:

```
public class PojoReleaseStrategy {
...
  public boolean canRelease(List<Long> numbers) {
    int sum = 0;
    for (long number: numbers) {
      sum += number;
    }
    return sum >= maxValue;
  }
}
```

Wherever it makes sense to do so, the release strategy method and the aggregator method can be combined into a single bean.

An implementation of the correlation strategy bean for the example above might be as follows:

```
public class PojoCorrelationStrategy {
...
  public Long groupNumbersByLastDigit(Long number) {
    return number % 10;
  }
}
```

The aggregator in the preceding example would group numbers by some criterion (in this case, the remainder after dividing by ten) and hold the group until the sum of the numbers provided by the payloads exceeds a certain value.

Wherever it makes sense to do so, the release strategy method, the correlation strategy method, and the aggregator method can be combined in a single bean. (Actually, all of them or any two of them can be combined.)

Aggregators and Spring Expression Language (SpEL)

Since Spring Integration 2.0, you can handle the various strategies (correlation, release, and aggregation) with SpEL, which we recommend if the logic behind such a release strategy is relatively simple. Suppose you have a legacy component that was designed to receive an array of objects. We know that the default release strategy assembles all aggregated messages in the `List`. Now we have two problems. First, we need to extract individual messages from the list. Second, we need to extract the payload of each message and assemble the array of objects. The following example solves both problems:

```
public String[] processRelease(List<Message<String>> messages){
    List<String> stringList = new ArrayList<String>();
    for (Message<String> message : messages) {
        stringList.add(message.getPayload());
    }
    return stringList.toArray(new String[]{});
}
```

However, with SpEL, such a requirement could actually be handled relatively easily with a one-line expression, thus sparing you from writing a custom class and configuring it as a bean. The following example shows how to do so:

```
<int:aggregator input-channel="aggChannel"
    output-channel="replyChannel"
    expression="#this.![payload].toArray()"/>
```

In the preceding configuration, we use a collection projection expression to assemble a new collection from the payloads of all the messages in the list and then transform it to an array, thus achieving the same result as the earlier Java code.

You can apply the same expression-based approach when dealing with custom release and correlation strategies.

Instead of defining a bean for a custom `CorrelationStrategy` in the `correlation-strategy` attribute, you can implement your simple correlation logic as a SpEL expression and configure it in the `correlation-strategy-expression` attribute, as the following example shows:

```
correlation-strategy-expression="payload.person.id"
```

In the preceding example, we assume that the payload has a `person` attribute with an `id`, which is going to be used to correlate messages.

Likewise, for the `ReleaseStrategy`, you can implement your release logic as a SpEL expression and configure it in the `release-strategy-expression` attribute. The root object for evaluation context is the `MessageGroup` itself. The `List` of messages can be referenced by using the `message` property of the group within the expression.

In releases prior to version 5.0, the root object was the collection of `Message<?>`, as the previous example shows:

```
release-strategy-expression="!messages.?[payload==5].empty"
```

In the preceding example, the root object of the SpEL evaluation context is the `MessageGroup` itself, and you are stating that, as soon as there is a message with payload of `5` in this group, the group should be released.

Aggregator and Group Timeout

Starting with version 4.0, two new mutually exclusive attributes have been introduced: `group-timeout` and `group-timeout-expression` (see the earlier description). See Configuring an Aggregator with XML. In some cases, you may need to emit the aggregator result (or discard the group) after a timeout if the `ReleaseStrategy` does not release when the current message arrives. For this purpose, the `groupTimeout` option lets scheduling the `MessageGroup` be forced to complete, as the following example shows:

```
<aggregator input-channel="input" output-channel="output"
        send-partial-result-on-expiry="true"
        group-timeout-expression="size() ge 2 ? 10000 : -1"
        release-strategy-expression="messages[0].headers.sequenceNumber ==
messages[0].headers.sequenceSize"/>
```

With this example, the normal release is possible if the aggregator receives the last message in sequence as defined by the `release-strategy-expression`. If that specific message does not arrive, the `groupTimeout` forces the group to complete after ten seconds, as long as the group contains at least two Messages.

The results of forcing the group to complete depends on the `ReleaseStrategy` and the `send-partial-result-on-expiry`. First, the release strategy is again consulted to see if a normal release is to be

made. While the group has not changed, the `ReleaseStrategy` can decide to release the group at this time. If the release strategy still does not release the group, it is expired. If `send-partial-result-on-expiry` is `true`, existing messages in the (partial) `MessageGroup` are released as a normal aggregator reply message to the `output-channel`. Otherwise, it is discarded.

There is a difference between `groupTimeout` behavior and `MessageGroupStoreReaper` (see Configuring an Aggregator with XML). The reaper initiates forced completion for all `MessageGroup` s in the `MessageGroupStore` periodically. The `groupTimeout` does it for each `MessageGroup` individually if a new message does not arrive during the `groupTimeout`. Also, the reaper can be used to remove empty groups (empty groups are retained in order to discard late messages if `expire-groups-upon-completion` is false).

Configuring an Aggregator with Annotations

The following example shows an aggregator configured with annotations:

```
public class Waiter {
  ...

  @Aggregator  ①
  public Delivery aggregatingMethod(List<OrderItem> items) {
    ...
  }

  @ReleaseStrategy  ②
  public boolean releaseChecker(List<Message<?>> messages) {
    ...
  }

  @CorrelationStrategy  ③
  public String correlateBy(OrderItem item) {
    ...
  }
}
```

① An annotation indicating that this method should be used as an aggregator. It must be specified if this class is used as an aggregator.

② An annotation indicating that this method is used as the release strategy of an aggregator. If not present on any method, the aggregator uses the `SimpleSequenceSizeReleaseStrategy`.

③ An annotation indicating that this method should be used as the correlation strategy of an aggregator. If no correlation strategy is indicated, the aggregator uses the `HeaderAttributeCorrelationStrategy` based on `CORRELATION_ID`.

All of the configuration options provided by the XML element are also available for the `@Aggregator` annotation.

The aggregator can be either referenced explicitly from XML or, if the `@MessageEndpoint` is defined

on the class, detected automatically through classpath scanning.

Annotation configuration (`@Aggregator` and others) for the Aggregator component covers only simple use cases, where most default options are sufficient. If you need more control over those options when using annotation configuration, consider using a `@Bean` definition for the `AggregatingMessageHandler` and mark its `@Bean` method with `@ServiceActivator`, as the following example shows:

```
@ServiceActivator(inputChannel = "aggregatorChannel")
@Bean
public MessageHandler aggregator(MessageGroupStore jdbcMessageGroupStore) {
     AggregatingMessageHandler aggregator =
                         new AggregatingMessageHandler(new
DefaultAggregatingMessageGroupProcessor(),
                                                       jdbcMessageGroupStore);
     aggregator.setOutputChannel(resultsChannel());
     aggregator.setGroupTimeoutExpression(new ValueExpression<>(500L));
     aggregator.setTaskScheduler(this.taskScheduler);
     return aggregator;
}
```

See Programming Model and Annotations on `@Bean` Methods for more information.

Starting with version 4.2, the `AggregatorFactoryBean` is available to simplify Java configuration for the `AggregatingMessageHandler`.

8.4.4. Managing State in an Aggregator: `MessageGroupStore`

Aggregator (and some other patterns in Spring Integration) is a stateful pattern that requires decisions to be made based on a group of messages that have arrived over a period of time, all with the same correlation key. The design of the interfaces in the stateful patterns (such as `ReleaseStrategy`) is driven by the principle that the components (whether defined by the framework or by a user) should be able to remain stateless. All state is carried by the `MessageGroup` and its management is delegated to the `MessageGroupStore`. The `MessageGroupStore` interface is defined as follows:

```
public interface MessageGroupStore {

    int getMessageCountForAllMessageGroups();

    int getMarkedMessageCountForAllMessageGroups();

    int getMessageGroupCount();

    MessageGroup getMessageGroup(Object groupId);

    MessageGroup addMessageToGroup(Object groupId, Message<?> message);

    MessageGroup markMessageGroup(MessageGroup group);

    MessageGroup removeMessageFromGroup(Object key, Message<?> messageToRemove);

    MessageGroup markMessageFromGroup(Object key, Message<?> messageToMark);

    void removeMessageGroup(Object groupId);

    void registerMessageGroupExpiryCallback(MessageGroupCallback callback);

    int expireMessageGroups(long timeout);
}
```

For more information, see the Javadoc.

The `MessageGroupStore` accumulates state information in `MessageGroups` while waiting for a release strategy to be triggered, and that event might not ever happen. So, to prevent stale messages from lingering, and for volatile stores to provide a hook for cleaning up when the application shuts down, the `MessageGroupStore` lets you register callbacks to apply to its `MessageGroups` when they expire. The interface is very straightforward, as the following listing shows:

```
public interface MessageGroupCallback {

    void execute(MessageGroupStore messageGroupStore, MessageGroup group);

}
```

The callback has direct access to the store and the message group so that it can manage the persistent state (for example, by entirely removing the group from the store).

The `MessageGroupStore` maintains a list of these callbacks, which it applies, on demand, to all messages whose timestamps are earlier than a time supplied as a parameter (see the `registerMessageGroupExpiryCallback(..)` and `expireMessageGroups(..)` methods, described earlier).

For more detail, see Managing State in an Aggregator: `MessageGroupStore`.

It is important not to use the same `MessageGroupStore` instance in different aggregator components, when you intend to rely on the `expireMessageGroups` functionality. Every `AbstractCorrelatingMessageHandler` registers its own `MessageGroupCallback` based on the `forceComplete()` callback. This way each group for expiration may be completed or discarded by the wrong aggregator. Starting with version 5.0.10, a `UniqueExpiryCallback` is used from the `AbstractCorrelatingMessageHandler` for the registration callback in the `MessageGroupStore`. The `MessageGroupStore`, in turn, checks for presence an instance of this class and logs an error with an appropriate message if one is already present in the callbacks set. This way the Framework disallows usage of the `MessageGroupStore` instance in different aggregators/resequencers to avoid the mentioned side effect of expiration the groups not created by the particular correlation handler.

You can call the `expireMessageGroups` method with a timeout value. Any message older than the current time minus this value is expired and has the callbacks applied. Thus, it is the user of the store that defines what is meant by message group “expiry”.

As a convenience for users, Spring Integration provides a wrapper for the message expiry in the form of a `MessageGroupStoreReaper`, as the following example shows:

```
<bean id="reaper" class="org...MessageGroupStoreReaper">
    <property name="messageGroupStore" ref="messageStore"/>
    <property name="timeout" value="30000"/>
</bean>

<task:scheduled-tasks scheduler="scheduler">
    <task:scheduled ref="reaper" method="run" fixed-rate="10000"/>
</task:scheduled-tasks>
```

The reaper is a `Runnable`. In the preceding example, the message group store’s expire method is called every ten seconds. The timeout itself is 30 seconds.

It is important to understand that the 'timeout' property of `MessageGroupStoreReaper` is an approximate value and is impacted by the the rate of the task scheduler, since this property is only checked on the next scheduled execution of the `MessageGroupStoreReaper` task. For example, if the timeout is set for ten minutes but the `MessageGroupStoreReaper` task is scheduled to run every hour and the last execution of the `MessageGroupStoreReaper` task happened one minute before the timeout, the `MessageGroup` does not expire for the next 59 minutes. Consequently, we recommend setting the rate to be at least equal to the value of the timeout or shorter.

In addition to the reaper, the expiry callbacks are invoked when the application shuts down

through a lifecycle callback in the `AbstractCorrelatingMessageHandler`.

The `AbstractCorrelatingMessageHandler` registers its own expiry callback, and this is the link with the boolean flag `send-partial-result-on-expiry` in the XML configuration of the aggregator. If the flag is set to `true`, then, when the expiry callback is invoked, any unmarked messages in groups that are not yet released can be sent on to the output channel.

When a shared `MessageStore` is used for different correlation endpoints, you must configure a proper `CorrelationStrategy` to ensure uniqueness for group IDs. Otherwise, unexpected behavior may happen when one correlation endpoint releases or expire messages from others. Messages with the same correlation key are stored in the same message group.

Some `MessageStore` implementations allow using the same physical resources, by partitioning the data. For example, the `JdbcMessageStore` has a `region` property, and the `MongoDbMessageStore` has a `collectionName` property.

For more information about the `MessageStore` interface and its implementations, see Message Store.

8.4.5. Flux Aggregator

In version 5.2, the `FluxAggregatorMessageHandler` component has been introduced. It is based on the Project Reactor `Flux.groupBy()` and `Flux.window()` operators. The incoming messages are emitted into the `FluxSink` initiated by the `Flux.create()` in the constructor of this component. If the `outputChannel` is not provided or it is not an instance of `ReactiveStreamsSubscribableChannel`, the subscription to the main `Flux` is done from the `Lifecycle.start()` implementation. Otherwise it is postponed to the subscription done by the `ReactiveStreamsSubscribableChannel` implementation. The messages are grouped by the `Flux.groupBy()` using a `CorrelationStrategy` for the group key. By default, the `IntegrationMessageHeaderAccessor.CORRELATION_ID` header of the message is consulted.

By default every closed window is released as a `Flux` in payload of a message to produce. This message contains all the headers from the first message in the window. This `Flux` in the output message payload must be subscribed and processed downstream. Such a logic can be customized (or superseded) by the `setCombineFunction(Function<Flux<Message<?>>, Mono<Message<?>>>)` configuration option of the `FluxAggregatorMessageHandler`. For example, if we would like to have a `List` of payloads in the final message, we can configure a `Flux.collectList()` like this:

```
fluxAggregatorMessageHandler.setCombineFunction(
                (messageFlux) ->
                        messageFlux
                                .map(Message::getPayload)
                                .collectList()
                                .map(GenericMessage::new));
```

There are several options in the `FluxAggregatorMessageHandler` to select an appropriate window

strategy:

- `setBoundaryTrigger(Predicate<Message<?>>)` - is propagated to the `Flux.windowUntil()` operator. See its JavaDocs for more information. Has a precedence over all other window options.
- `setWindowSize(int)` and `setWindowSizeFunction(Function<Message<?>, Integer>)` - is propagated to the `Flux.window(int)` or `windowTimeout(int, Duration)`. By default a window size is calculated from the first message in group and its `IntegrationMessageHeaderAccessor.SEQUENCE_SIZE` header.
- `setWindowTimespan(Duration)` - is propagated to the `Flux.window(Duration)` or `windowTimeout(int, Duration)` depending in the window size configuration.
- `setWindowConfigurer(Function<Flux<Message<?>>, Flux<Flux<Message<?>>>>)` - a function to apply a transformation into the grouped fluxes for any custom window operation not covered by the exposed options.

Since this component is a `MessageHandler` implementation it can simply be used as a `@Bean` definition together with a `@ServiceActivator` messaging annotation. With Java DSL it can be used from the `.handle()` EIP-method. The sample below demonstrates how we can register an `IntegrationFlow` at runtime and how a `FluxAggregatorMessageHandler` can be correlated with a splitter upstream:

```
IntegrationFlow fluxFlow =
        (flow) -> flow
                .split()
                .channel(MessageChannels.flux())
                .handle(new FluxAggregatorMessageHandler());

IntegrationFlowContext.IntegrationFlowRegistration registration =
        this.integrationFlowContext.registration(fluxFlow)
                .register();

@SuppressWarnings("unchecked")
Flux<Message<?>> window =
        registration.getMessagingTemplate()
                .convertSendAndReceive(new Integer[] { 0, 1, 2, 3, 4, 5, 6, 7, 8,
9 }, Flux.class);
```

8.5. Resequencer

The resequencer is related to the aggregator but serves a different purpose. While the aggregator combines messages, the resequencer passes messages through without changing them.

8.5.1. Functionality

The resequencer works in a similar way to the aggregator, in the sense that it uses the `CORRELATION_ID` to store messages in groups. The difference is that the Resequencer does not process the messages in any way. Instead, it releases them in the order of their `SEQUENCE_NUMBER` header values.

With respect to that, you can opt to release all messages at once (after the whole sequence, according to the `SEQUENCE_SIZE`, and other possibilities) or as soon as a valid sequence is available. (We cover what we mean by "a valid sequence" later in this chapter.)

The resequencer is intended to resequence relatively short sequences of messages with small gaps. If you have a large number of disjoint sequences with many gaps, you may experience performance issues.

8.5.2. Configuring a Resequencer

See Aggregators and Resequencers for configuring a resequencer in Java DSL.

Configuring a resequencer requires only including the appropriate element in XML.

The following example shows a resequencer configuration:

```
<int:channel id="inputChannel"/>

<int:channel id="outputChannel"/>

<int:resequencer id="completelyDefinedResequencer"  ①
  input-channel="inputChannel"  ②
  output-channel="outputChannel"  ③
  discard-channel="discardChannel"  ④
  release-partial-sequences="true"  ⑤
  message-store="messageStore"  ⑥
  send-partial-result-on-expiry="true"  ⑦
  send-timeout="86420000"  ⑧
  correlation-strategy="correlationStrategyBean"  ⑨
  correlation-strategy-method="correlate"  ⑩
  correlation-strategy-expression="headers['something']"  ⑪
  release-strategy="releaseStrategyBean"  ⑫
  release-strategy-method="release"  ⑬
  release-strategy-expression="size() == 10"  ⑭
  empty-group-min-timeout="60000"  ⑮

  lock-registry="lockRegistry"  ⑯

  group-timeout="60000"  ⑰
  group-timeout-expression="size() ge 2 ? 100 : -1"  ⑱
  scheduler="taskScheduler" />  ⑲
  expire-group-upon-timeout="false" />  ⑳
```

① The id of the resequencer is optional.

② The input channel of the resequencer. Required.

③ The channel to which the resequencer sends the reordered messages. Optional.

④ The channel to which the resequencer sends the messages that timed out (if `send-partial-result-on-timeout` is set to `false`). Optional.

⑤ Whether to send out ordered sequences as soon as they are available or only after the whole message group arrives. Optional. (The default is `false`.)

⑥ A reference to a `MessageGroupStore` that can be used to store groups of messages under their correlation key until they are complete. Optional. (The default is a volatile in-memory store.)

⑦ Whether, upon the expiration of the group, the ordered group should be sent out (even if some of the messages are missing). Optional. (The default is false.) See Managing State in an Aggregator: `MessageGroupStore`.

⑧ The timeout interval to wait when sending a reply `Message` to the `output-channel` or `discard-channel`. Defaults to `-1`, which blocks indefinitely. It is applied only if the output channel has some 'sending' limitations, such as a `QueueChannel` with a fixed 'capacity'. In this case, a `MessageDeliveryException` is thrown. The `send-timeout` is ignored for `AbstractSubscribableChannel` implementations. For `group-timeout(-expression)`, the

`MessageDeliveryException` from the scheduled expire task leads this task to be rescheduled. Optional.

⑨ A reference to a bean that implements the message correlation (grouping) algorithm. The bean can be an implementation of the `CorrelationStrategy` interface or a POJO. In the latter case, the `correlation-strategy-method` attribute must also be defined. Optional. (By default, the aggregator uses the `IntegrationMessageHeaderAccessor.CORRELATION_ID` header.)

⑩ A method that is defined on the bean referenced by `correlation-strategy` and that implements the correlation decision algorithm. Optional, with restrictions (requires `correlation-strategy` to be present).

⑪ A SpEL expression representing the correlation strategy. Example: `"headers['something']"`. Only one of `correlation-strategy` or `correlation-strategy-expression` is allowed.

⑫ A reference to a bean that implements the release strategy. The bean can be an implementation of the `ReleaseStrategy` interface or a POJO. In the latter case, the `release-strategy-method` attribute must also be defined. Optional (by default, the aggregator will use the `IntegrationMessageHeaderAccessor.SEQUENCE_SIZE` header attribute).

⑬ A method that is defined on the bean referenced by `release-strategy` and that implements the completion decision algorithm. Optional, with restrictions (requires `release-strategy` to be present).

⑭ A SpEL expression representing the release strategy. The root object for the expression is a `MessageGroup`. Example: `"size() == 5"`. Only one of `release-strategy` or `release-strategy-expression` is allowed.

⑮ Only applies if a `MessageGroupStoreReaper` is configured for the `<resequencer>` `MessageStore`. By default, when a `MessageGroupStoreReaper` is configured to expire partial groups, empty groups are also removed. Empty groups exist after a group is released normally. This is to enable the detection and discarding of late-arriving messages. If you wish to expire empty groups on a longer schedule than expiring partial groups, set this property. Empty groups are then not removed from the `MessageStore` until they have not been modified for at least this number of milliseconds. Note that the actual time to expire an empty group is also affected by the reaper's timeout property, and it could be as much as this value plus the timeout.

⑯ See Configuring an Aggregator with XML.

⑰ See Configuring an Aggregator with XML.

⑱ See Configuring an Aggregator with XML.

⑲ See Configuring an Aggregator with XML.

⑳ By default, when a group is completed due to a timeout (or by a `MessageGroupStoreReaper`), the empty group's metadata is retained. Late arriving messages are immediately discarded. Set this to `true` to remove the group completely. Then, late arriving messages start a new group and are not be discarded until the group again times out. The new group is never released normally because of the "hole" in the sequence range that caused the timeout. Empty groups can be expired (completely removed) later by using a `MessageGroupStoreReaper` together with the `empty-group-min-timeout` attribute. Starting with version 5.0, empty groups are also scheduled for removal after the `empty-group-min-timeout` elapses. The default is 'false'.

Since there is no custom behavior to be implemented in Java classes for resequencers, there is no annotation support for it.

8.6. Message Handler Chain

The `MessageHandlerChain` is an implementation of `MessageHandler` that can be configured as a single message endpoint while actually delegating to a chain of other handlers, such as filters, transformers, splitters, and so on. When several handlers need to be connected in a fixed, linear progression, this can lead to a much simpler configuration. For example, it is fairly common to provide a transformer before other components. Similarly, when you provide a filter before some other component in a chain, you essentially create a selective consumer. In either case, the chain requires only a single `input-channel` and a single `output-channel`, eliminating the need to define channels for each individual component.

Spring Integration's `Filter` provides a boolean property: `throwExceptionOnRejection`. When you provide multiple selective consumers on the same point-to-point channel with different acceptance criteria, you should set this value 'true' (the default is `false`) so that the dispatcher knows that the message was rejected and, as a result, tries to pass the message on to other subscribers. If the exception were not thrown, it would appear to the dispatcher that the message had been passed on successfully even though the filter had dropped the message to prevent further processing. If you do indeed want to "drop" the messages, the filter's 'discard-channel' might be useful, since it does give you a chance to perform some operation with the dropped message (such as sending it to a JMS queue or writing it to a log).

The handler chain simplifies configuration while internally maintaining the same degree of loose coupling between components, and it is trivial to modify the configuration if at some point a non-linear arrangement is required.

Internally, the chain is expanded into a linear setup of the listed endpoints, separated by anonymous channels. The reply channel header is not taken into account within the chain. Only after the last handler is invoked is the resulting message forwarded to the reply channel or the chain's output channel. Because of this setup, all handlers except the last must implement the `MessageProducer` interface (which provides a 'setOutputChannel()' method). If the `outputChannel` on the `MessageHandlerChain` is set, the last handler needs only an output channel.

As with other endpoints, the `output-channel` is optional. If there is a reply message at the end of the chain, the output-channel takes precedence. However, if it is not available, the chain handler checks for a reply channel header on the inbound message as a fallback.

In most cases, you need not implement `MessageHandler` yourself. The next section focuses on namespace support for the chain element. Most Spring Integration endpoints, such as service activators and transformers, are suitable for use within a `MessageHandlerChain`.

8.6.1. Configuring a Chain

The <chain> element provides an input-channel attribute. If the last element in the chain is capable of producing reply messages (optional), it also supports an output-channel attribute. The sub-elements are then filters, transformers, splitters, and service-activators. The last element may also be a router or an outbound channel adapter. The following example shows a chain definition:

```
<int:chain input-channel="input" output-channel="output">
    <int:filter ref="someSelector" throw-exception-on-rejection="true"/>
    <int:header-enricher>
        <int:header name="thing1" value="thing2"/>
    </int:header-enricher>
    <int:service-activator ref="someService" method="someMethod"/>
</int:chain>
```

The <header-enricher> element used in the preceding example sets a message header named thing1 with a value of thing2 on the message. A header enricher is a specialization of Transformer that touches only header values. You could obtain the same result by implementing a MessageHandler that did the header modifications and wiring that as a bean, but the header-enricher is a simpler option.

The <chain> can be configured as the last 'black-box' consumer of the message flow. For this solution, you can to put it at the end of the <chain> some <outbound-channel-adapter>, as the following example shows:

```
<int:chain input-channel="input">
    <int-xml:marshalling-transformer marshaller="marshaller" result-type=
"StringResult" />
    <int:service-activator ref="someService" method="someMethod"/>
    <int:header-enricher>
        <int:header name="thing1" value="thing2"/>
    </int:header-enricher>
    <int:logging-channel-adapter level="INFO" log-full-message="true"/>
</int:chain>
```

Disallowed Attributes and Elements

Certain attributes, such as `order` and `input-channel` are not allowed to be specified on components used within a chain. The same is true for the poller sub-element.

For the Spring Integration core components, the XML schema itself enforces some of these constraints. However, for non-core components or your own custom components, these constraints are enforced by the XML namespace parser, not by the XML schema.

These XML namespace parser constraints were added with Spring Integration 2.2. If you try to use disallowed attributes and elements, the XML namespace parser throws a `BeanDefinitionParsingException`.

8.6.2. Using the 'id' Attribute

Beginning with Spring Integration 3.0, if a chain element is given an `id` attribute, the bean name for the element is a combination of the chain's `id` and the `id` of the element itself. Elements without `id` attributes are not registered as beans, but each one is given a `componentName` that includes the chain `id`. Consider the following example:

```
<int:chain id="somethingChain" input-channel="input">
    <int:service-activator id="somethingService" ref="someService" method=
"someMethod"/>
    <int:object-to-json-transformer/>
</int:chain>
```

In the preceding example:

- The `<chain>` root element has an `id` of 'somethingChain'. Consequently, the `AbstractEndpoint` implementation (`PollingConsumer` or `EventDrivenConsumer`, depending on the `input-channel` type) bean takes this value as its bean name.
- The `MessageHandlerChain` bean acquires a bean alias ('somethingChain.handler'), which allows direct access to this bean from the `BeanFactory`.
- The `<service-activator>` is not a fully fledged messaging endpoint (it is not a `PollingConsumer` or `EventDrivenConsumer`). It is a `MessageHandler` within the `<chain>`. In this case, the bean name registered with the `BeanFactory` is 'somethingChain$child.somethingService.handler'.
- The `componentName` of this `ServiceActivatingHandler` takes the same value but without the '.handler' suffix. It becomes 'somethingChain$child.somethingService'.
- The last `<chain>` sub-component, `<object-to-json-transformer>`, does not have an `id` attribute. Its `componentName` is based on its position in the `<chain>`. In this case, it is 'somethingChain$child#1'. (The final element of the name is the order within the chain, beginning with '#0'). Note, this transformer is not registered as a bean within the application context, so it does not get a `beanName`. However its `componentName` has a value that is useful for logging and other purposes.

The id attribute for <chain> elements lets them be eligible for JMX export, and they are trackable in the message history. You can access them from the BeanFactory by using the appropriate bean name, as discussed earlier.

It is useful to provide an explicit id attribute on <chain> elements to simplify the identification of sub-components in logs and to provide access to them from the BeanFactory etc.

8.6.3. Calling a Chain from within a Chain

Sometimes, you need to make a nested call to another chain from within a chain and then come back and continue execution within the original chain. To accomplish this, you can use a messaging gateway by including a <gateway> element, as the following example shows:

```
<int:chain id="main-chain" input-channel="in" output-channel="out">
    <int:header-enricher>
      <int:header name="name" value="Many" />
    </int:header-enricher>
    <int:service-activator>
      <bean class="org.foo.SampleService" />
    </int:service-activator>
    <int:gateway request-channel="inputA"/>
</int:chain>

<int:chain id="nested-chain-a" input-channel="inputA">
    <int:header-enricher>
        <int:header name="name" value="Moe" />
    </int:header-enricher>
    <int:gateway request-channel="inputB"/>
    <int:service-activator>
        <bean class="org.foo.SampleService" />
    </int:service-activator>
</int:chain>

<int:chain id="nested-chain-b" input-channel="inputB">
    <int:header-enricher>
        <int:header name="name" value="Jack" />
    </int:header-enricher>
    <int:service-activator>
        <bean class="org.foo.SampleService" />
    </int:service-activator>
</int:chain>
```

In the preceding example, nested-chain-a is called at the end of main-chain processing by the 'gateway' element configured there. While in nested-chain-a, a call to a nested-chain-b is made after header enrichment. Then the flow comes back to finish execution in nested-chain-b. Finally, the flow returns to main-chain. When the nested version of a <gateway> element is defined in the chain,

it does not require the `service-interface` attribute. Instead, it takes the message in its current state and places it on the channel defined in the `request-channel` attribute. When the downstream flow initiated by that gateway completes, a `Message` is returned to the gateway and continues its journey within the current chain.

8.7. Scatter-Gather

Starting with version 4.1, Spring Integration provides an implementation of the scatter-gather enterprise integration pattern. It is a compound endpoint for which the goal is to send a message to the recipients and aggregate the results. As noted in *Enterprise Integration Patterns*, it is a component for scenarios such as "best quote", where we need to request information from several suppliers and decide which one provides us with the best term for the requested item.

Previously, the pattern could be configured by using discrete components. This enhancement brings more convenient configuration.

The `ScatterGatherHandler` is a request-reply endpoint that combines a `PublishSubscribeChannel` (or a `RecipientListRouter`) and an `AggregatingMessageHandler`. The request message is sent to the `scatter` channel, and the `ScatterGatherHandler` waits for the reply that the aggregator sends to the `outputChannel`.

8.7.1. Functionality

The `Scatter-Gather` pattern suggests two scenarios: "auction" and "distribution". In both cases, the `aggregation` function is the same and provides all the options available for the `AggregatingMessageHandler`. (Actually, the `ScatterGatherHandler` requires only an `AggregatingMessageHandler` as a constructor argument.) See Aggregator for more information.

Auction

The auction `Scatter-Gather` variant uses "publish-subscribe" logic for the request message, where the "scatter" channel is a `PublishSubscribeChannel` with `apply-sequence="true"`. However, this channel can be any `MessageChannel` implementation (as is the case with the `request-channel` in the `ContentEnricher` — see Content Enricher). However, in this case, you should create your own custom `correlationStrategy` for the `aggregation` function.

Distribution

The distribution `Scatter-Gather` variant is based on the `RecipientListRouter` (see `RecipientListRouter`) with all available options for the `RecipientListRouter`. This is the second `ScatterGatherHandler` constructor argument. If you want to rely on only the default `correlationStrategy` for the `recipient-list-router` and the `aggregator`, you should specify `apply-sequence="true"`. Otherwise, you should supply a custom `correlationStrategy` for the `aggregator`. Unlike the `PublishSubscribeChannel` variant (the auction variant), having a `recipient-list-router` `selector` option lets filter target suppliers based on the message. With `apply-sequence="true"`, the default `sequenceSize` is supplied, and the `aggregator` can release the group correctly. The distribution option is mutually exclusive with the auction option.

For both the auction and the distribution variants, the request (scatter) message is enriched with

the gatherResultChannel header to wait for a reply message from the aggregator.

By default, all suppliers should send their result to the replyChannel header (usually by omitting the output-channel from the ultimate endpoint). However, the gatherChannel option is also provided, letting suppliers send their reply to that channel for the aggregation.

8.7.2. Configuring a Scatter-Gather Endpoint

The following example shows Java configuration for the bean definition for Scatter-Gather:

```
@Bean
public MessageHandler distributor() {
    RecipientListRouter router = new RecipientListRouter();
    router.setApplySequence(true);
    router.setChannels(Arrays.asList(distributionChannel1(), distributionChannel2
(),
            distributionChannel3()));
    return router;
}

@Bean
public MessageHandler gatherer() {
    return new AggregatingMessageHandler(
            new ExpressionEvaluatingMessageGroupProcessor("^[payload gt 5] ?: -1D
"),
            new SimpleMessageStore(),
            new HeaderAttributeCorrelationStrategy(
                    IntegrationMessageHeaderAccessor.CORRELATION_ID),
            new ExpressionEvaluatingReleaseStrategy("size() == 2"));
}

@Bean
@ServiceActivator(inputChannel = "distributionChannel")
public MessageHandler scatterGatherDistribution() {
    ScatterGatherHandler handler = new ScatterGatherHandler(distributor(),
gatherer());
    handler.setOutputChannel(output());
    return handler;
}
```

In the preceding example, we configure the RecipientListRouter distributor bean with applySequence="true" and the list of recipient channels. The next bean is for an AggregatingMessageHandler. Finally, we inject both those beans into the ScatterGatherHandler bean definition and mark it as a @ServiceActivator to wire the scatter-gather component into the integration flow.

The following example shows how to configure the <scatter-gather> endpoint by using the XML namespace:

```
<scatter-gather
        id=""  ①
        auto-startup=""  ②
        input-channel=""  ③
        output-channel=""  ④
        scatter-channel=""  ⑤
        gather-channel=""  ⑥
        order=""  ⑦
        phase=""  ⑧
        send-timeout=""  ⑨
        gather-timeout=""  ⑩
        requires-reply="" > ⑪
            <scatterer/>  ⑫
            <gatherer/>  ⑬
</scatter-gather>
```

① The id of the endpoint. The `ScatterGatherHandler` bean is registered with an alias of `id + '.handler'`. The `RecipientListRouter` bean is registered with an alias of `id + '.scatterer'`. The `AggregatingMessageHandler`bean is registered with an alias of `id + '.gatherer'`. Optional. (The `BeanFactory` generates a default `id` value.)

② Lifecycle attribute signaling whether the endpoint should be started during application context initialization. In addition, the `ScatterGatherHandler` also implements `Lifecycle` and starts and stops `gatherEndpoint`, which is created internally if a `gather-channel` is provided. Optional. (The default is `true`.)

③ The channel on which to receive request messages to handle them in the `ScatterGatherHandler`. Required.

④ The channel to which the `ScatterGatherHandler` sends the aggregation results. Optional. (Incoming messages can specify a reply channel themselves in the `replyChannel` message header).

⑤ The channel to which to send the scatter message for the auction scenario. Optional. Mutually exclusive with the `<scatterer>` sub-element.

⑥ The channel on which to receive replies from each supplier for the aggregation. It is used as the `replyChannel` header in the scatter message. Optional. By default, the `FixedSubscriberChannel` is created.

⑦ The order of this component when more than one handler is subscribed to the same `DirectChannel` (use for load balancing purposes). Optional.

⑧ Specifies the phase in which the endpoint should be started and stopped. The startup order proceeds from lowest to highest, and the shutdown order is from highest to lowest. By default, this value is `Integer.MAX_VALUE`, meaning that this container starts as late as possible and stops as soon as possible. Optional.

⑨ The timeout interval to wait when sending a reply `Message` to the `output-channel`. By default, the send blocks for one second. It applies only if the output channel has some 'sending' limitations — for example, a `QueueChannel` with a fixed 'capacity' that is full. In this case, a `MessageDeliveryException` is thrown. The `send-timeout` is ignored for

`AbstractSubscribableChannel` implementations. For `group-timeout(-expression)`, the `MessageDeliveryException` from the scheduled expire task leads this task to be rescheduled. Optional.

⑩ Lets you specify how long the scatter-gather waits for the reply message before returning. By default, it waits indefinitely. 'null' is returned if the reply times out. Optional. It defaults to `-1`, meaning to wait indefinitely.

⑪ Specifies whether the scatter-gather must return a non-null value. This value is `true` by default. Consequently, a `ReplyRequiredException` is thrown when the underlying aggregator returns a null value after `gather-timeout`. Note, if `null` is a possibility, the `gather-timeout` should be specified to avoid an indefinite wait.

⑫ The `<recipient-list-router>` options. Optional. Mutually exclusive with `scatter-channel` attribute.

⑬ The `<aggregator>` options. Required.

8.7.3. Error Handling

Since Scatter-Gather is a multi request-reply component, error handling has some extra complexity. In some cases, it is better to just catch and ignore downstream exceptions if the `ReleaseStrategy` allows the process to finish with fewer replies than requests. In other cases something like a "compensation message" should be considered for returning from sub-flow, when an error happens.

Every async sub-flow should be configured with a `errorChannel` header for the proper error message sending from the `MessagePublishingErrorHandler`. Otherwise, an error will be sent to the global `errorChannel` with the common error handling logic. See Error Handling for more information about async error processing.

Synchronous flows may use an `ExpressionEvaluatingRequestHandlerAdvice` for ignoring the exception or returning a compensation message. When an exception is thrown from one of the sub-flows to the `ScatterGatherHandler`, it is just re-thrown to upstream. This way all other sub-flows will work for nothing and their replies are going to be ignored in the `ScatterGatherHandler`. This might be an expected behavior sometimes, but in most cases it would be better to handle the error in the particular sub-flow without impacting all others and the expectations in the gatherer.

Starting with version 5.1.3, the `ScatterGatherHandler` is supplied with the `errorChannelName` option. It is populated to the `errorChannel` header of the scatter message and is used in the when async error happens or can be used in the regular synchronous sub-flow for directly sending an error message.

The sample configuration below demonstrates async error handling by returning a compensation message:

```
@Bean
public IntegrationFlow scatterGatherAndExecutorChannelSubFlow(TaskExecutor
taskExecutor) {
    return f -> f
            .scatterGather(
                    scatterer -> scatterer
                            .applySequence(true)
                            .recipientFlow(f1 -> f1.transform(p -> "Sub-flow#1"))
                            .recipientFlow(f2 -> f2
                                    .channel(c -> c.executor(taskExecutor))
                                    .transform(p -> {
                                        throw new RuntimeException("Sub-flow#2");
                                    })),
                    null,
                    s -> s.errorChannel("scatterGatherErrorChannel"));
}

@ServiceActivator(inputChannel = "scatterGatherErrorChannel")
public Message<?> processAsyncScatterError(MessagingException payload) {
    return MessageBuilder.withPayload(payload.getCause().getCause())
            .copyHeaders(payload.getFailedMessage().getHeaders())
            .build();
}
```

To produce a proper reply, we have to copy headers (including `replyChannel` and `errorChannel`) from the `failedMessage` of the `MessagingException` that has been sent to the `scatterGatherErrorChannel` by the `MessagePublishingErrorHandler`. This way the target exception is returned to the gatherer of the `ScatterGatherHandler` for reply messages group completion. Such an exception `payload` can be filtered out in the `MessageGroupProcessor` of the gatherer or processed other way downstream, after the scatter-gather endpoint.

Before sending scattering results to the gatherer, `ScatterGatherHandler` reinstates the request message headers, including reply and error channels if any. This way errors from the `AggregatingMessageHandler` are going to be propagated to the caller, even if an async hand off is applied in scatter recipient subflows. For successful operation, a `gatherResultChannel`, `originalReplyChannel` and `originalErrorChannel` headers must be transferred back to replies from scatter recipient subflows. In this case a reasonable, finite `gatherTimeout` must be configured for the `ScatterGatherHandler`. Otherwise it is going to be blocked waiting for a reply from the gatherer forever, by default.

8.8. Thread Barrier

Sometimes, we need to suspend a message flow thread until some other asynchronous event occurs. For example, consider an HTTP request that publishes a message to RabbitMQ. We might wish to not reply to the user until the RabbitMQ broker has issued an acknowledgment that the

message was received.

In version 4.2, Spring Integration introduced the `<barrier/>` component for this purpose. The underlying `MessageHandler` is the `BarrierMessageHandler`. This class also implements `MessageTriggerAction`, in which a message passed to the `trigger()` method releases a corresponding thread in the `handleRequestMessage()` method (if present).

The suspended thread and trigger thread are correlated by invoking a `CorrelationStrategy` on the messages. When a message is sent to the `input-channel`, the thread is suspended for up to `timeout` milliseconds, waiting for a corresponding trigger message. The default correlation strategy uses the `IntegrationMessageHeaderAccessor.CORRELATION_ID` header. When a trigger message arrives with the same correlation, the thread is released. The message sent to the `output-channel` after release is constructed by using a `MessageGroupProcessor`. By default, the message is a `Collection<?>` of the two payloads, and the headers are merged by using a `DefaultAggregatingMessageGroupProcessor`.

If the `trigger()` method is invoked first (or after the main thread times out), it is suspended for up to `timeout` waiting for the suspending message to arrive. If you do not want to suspend the trigger thread, consider handing off to a `TaskExecutor` instead so that its thread is suspended instead.

The `requires-reply` property determines the action to take if the suspended thread times out before the trigger message arrives. By default, it is `false`, which means the endpoint returns `null`, the flow ends, and the thread returns to the caller. When `true`, a `ReplyRequiredException` is thrown.

You can call the `trigger()` method programmatically (obtain the bean reference by using the name, `barrier.handler` — where `barrier` is the bean name of the barrier endpoint). Alternatively, you can configure an `<outbound-channel-adapter/>` to trigger the release.

Only one thread can be suspended with the same correlation. The same correlation can be used multiple times but only once concurrently. An exception is thrown if a second thread arrives with the same correlation.

The following example shows how to use a custom header for correlation:

```
<int:barrier id="barrier1" input-channel="in" output-channel="out"
        correlation-strategy-expression="headers['myHeader']"
        output-processor="myOutputProcessor"
        discard-channel="lateTriggerChannel"
        timeout="10000">
</int:barrier>

<int:outbound-channel-adapter channel="release" ref="barrier1.handler" method="
trigger" />
```

Depending on which one has a message arrive first, either the thread sending a message to `in` or the thread sending a message to `release` waits for up to ten seconds until the other message arrives. When the message is released, the `out` channel is sent a message that combines the result of invoking the custom `MessageGroupProcessor` bean, named `myOutputProcessor`. If the main thread

times out and a trigger arrives later, you can configure a discard channel to which the late trigger is sent. The following example shows the Java configuration to do so:

```
@Configuration
@EnableIntegration
public class Config {

    @ServiceActivator(inputChannel="in")
    @Bean
    public BarrierMessageHandler barrier() {
        BarrierMessageHandler barrier = new BarrierMessageHandler(10000);
        barrier.setOutputChannel(out());
        barrier.setDiscardChannel(lateTriggers());
        return barrier;
    }

    @ServiceActivator (inputChannel="release")
    @Bean
    public MessageHandler releaser() {
        return new MessageHandler() {

            @Override
            public void handleMessage(Message<?> message) throws MessagingException {
                barrier().trigger(message);
            }

        };
    }

}
```

For an example of this component, see the barrier sample application.

Chapter 9. Message Transformation

9.1. Transformer

Message transformers play a very important role in enabling the loose-coupling of message producers and message consumers. Rather than requiring every message-producing component to know what type is expected by the next consumer, you can add transformers between those components. Generic transformers, such as one that converts a `String` to an XML Document, are also highly reusable.

For some systems, it may be best to provide a canonical data model, but Spring Integration's general philosophy is not to require any particular format. Rather, for maximum flexibility, Spring Integration aims to provide the simplest possible model for extension. As with the other endpoint types, the use of declarative configuration in XML or Java annotations enables simple POJOs to be adapted for the role of message transformers. The rest of this chapter describes these configuration options.

For the sake of maximizing flexibility, Spring does not require XML-based message payloads. Nevertheless, the framework does provide some convenient transformers for dealing with XML-based payloads if that is indeed the right choice for your application. For more information on those transformers, see XML Support - Dealing with XML Payloads.

9.1.1. Configuring a Transformer with XML

The `<transformer>` element is used to create a message-transforming endpoint. In addition to `input-channel` and `output-channel` attributes, it requires a ` attribute`. The `ref` may either point to an object that contains the `@Transformer` annotation on a single method (see Configuring a Transformer with Annotations), or it may be combined with an explicit method name value provided in the `method` attribute.

```
<int:transformer id="testTransformer" ref="testTransformerBean" input-channel=
"inChannel"
             method="transform" output-channel="outChannel"/>
<beans:bean id="testTransformerBean" class="org.foo.TestTransformer" />
```

Using a `ref` attribute is generally recommended if the custom transformer handler implementation can be reused in other `<transformer>` definitions. However, if the custom transformer handler implementation should be scoped to a single definition of the `<transformer>`, you can define an inner bean definition, as the following example shows:

```
<int:transformer id="testTransformer" input-channel="inChannel" method="transform"
                 output-channel="outChannel">
  <beans:bean class="org.foo.TestTransformer"/>
</transformer>
```

Using both the `ref` attribute and an inner handler definition in the same `<transformer>` configuration is not allowed, as it creates an ambiguous condition and results in an exception being thrown.

If the `ref` attribute references a bean that extends `AbstractMessageProducingHandler` (such as transformers provided by the framework itself), the configuration is optimized by injecting the output channel into the handler directly. In this case, each `ref` must be to a separate bean instance (or a `prototype`-scoped bean) or use the inner `<bean/>` configuration type. If you inadvertently reference the same message handler from multiple beans, you get a configuration exception.

When using a POJO, the method that is used for transformation may expect either the `Message` type or the payload type of inbound messages. It may also accept message header values either individually or as a full map by using the `@Header` and `@Headers` parameter annotations, respectively. The return value of the method can be any type. If the return value is itself a `Message`, that is passed along to the transformer's output channel.

As of Spring Integration 2.0, a message transformer's transformation method can no longer return `null`. Returning `null` results in an exception, because a message transformer should always be expected to transform each source message into a valid target message. In other words, a message transformer should not be used as a message filter, because there is a dedicated `<filter>` option for that. However, if you do need this type of behavior (where a component might return `null` and that should not be considered an error), you could use a service activator. Its `requires-reply` value is `false` by default, but that can be set to `true` in order to have exceptions thrown for `null` return values, as with the transformer.

9.1.2. Transformers and Spring Expression Language (SpEL)

Like routers, aggregators, and other components, as of Spring Integration 2.0, transformers can also benefit from SpEL support whenever transformation logic is relatively simple. The following example shows how to use a SpEL expression:

```
<int:transformer input-channel="inChannel"
    output-channel="outChannel"
    expression="payload.toUpperCase() + '- [' +
T(java.lang.System).currentTimeMillis() + ']'"/>
```

The preceding example transforms the payload without writing a custom transformer. Our payload

(assumed to be a `String`) is upper-cased, concatenated with the current timestamp, and has some formatting applied.

9.1.3. Common Transformers

Spring Integration provides a few transformer implementations.

Object-to-String Transformer

Because it is fairly common to use the `toString()` representation of an `Object`, Spring Integration provides an `ObjectToStringTransformer` whose output is a `Message` with a String `payload`. That `String` is the result of invoking the `toString()` operation on the inbound Message's payload. The following example shows how to declare an instance of the object-to-string transformer:

```
<int:object-to-string-transformer input-channel="in" output-channel="out"/>
```

A potential use for this transformer would be sending some arbitrary object to the 'outbound-channel-adapter' in the `file` namespace. Whereas that channel adapter only supports `String`, byte-array, or `java.io.File` payloads by default, adding this transformer immediately before the adapter handles the necessary conversion. That works fine as long as the result of the `toString()` call is what you want to be written to the file. Otherwise, you can provide a custom POJO-based transformer by using the generic 'transformer' element shown previously.

When debugging, this transformer is not typically necessary, since the 'logging-channel-adapter' is capable of logging the message payload. See Wire Tap for more detail.

The object-to-string transformer is very simple. It invokes `toString()` on the inbound payload. Since Spring Integration 3.0, there are two exceptions to this rule:

- If the payload is a `char[]`, it invokes `new String(payload)`.
- If the payload is a `byte[]`, it invokes `new String(payload, charset)`, where `charset` is UTF-8 by default. The `charset` can be modified by supplying the charset attribute on the transformer.

For more sophistication (such as selection of the charset dynamically, at runtime), you can use a SpEL expression-based transformer instead, as the following example shows:

```
<int:transformer input-channel="in" output-channel="out"
       expression="new java.lang.String(payload, headers['myCharset']"
/>
```

If you need to serialize an `Object` to a byte array or deserialize a byte array back into an `Object`,

Spring Integration provides symmetrical serialization transformers. These use standard Java serialization by default, but you can provide an implementation of Spring 3.0's serializer or seserializer strategies by using the 'serializer' and 'deserializer' attributes, respectively. The following example shows to use Spring's serializer and deserializer:

```
<int:payload-serializing-transformer input-channel="objectsIn" output-channel=
"bytesOut"/>

<int:payload-deserializing-transformer input-channel="bytesIn" output-channel=
"objectsOut"
    white-list="com.mycom.*,com.yourcom.*"/>
```

When deserializing data from untrusted sources, you should consider adding a `white-list` of package and class patterns. By default, all classes are deserialized.

`Object`-to-`Map` and `Map`-to-`Object` Transformers

Spring Integration also provides `Object`-to-`Map` and `Map`-to-`Object` transformers, which use the JSON to serialize and de-serialize the object graphs. The object hierarchy is introspected to the most primitive types (`String`, `int`, and so on). The path to this type is described with SpEL, which becomes the `key` in the transformed `Map`. The primitive type becomes the value.

Consider the following example:

```
public class Parent{
    private Child child;
    private String name;
    // setters and getters are omitted
}

public class Child{
    private String name;
    private List<String> nickNames;
    // setters and getters are omitted
}
```

The two classes in the preceding example are transformed to the following `Map`:

```
{person.name=George, person.child.name=Jenna, person.child.nickNames[0]=Jen ...}
```

The JSON-based `Map` lets you describe the object structure without sharing the actual types, which lets you restore and rebuild the object graph into a differently typed object graph, as long as you

maintain the structure.

For example, the preceding structure could be restored back to the following object graph by using the `Map`-to-`Object` transformer:

```
public class Father {
    private Kid child;
    private String name;
    // setters and getters are omitted
}

public class Kid {
    private String name;
    private List<String> nickNames;
    // setters and getters are omitted
}
```

If you need to create a “structured” map, you can provide the 'flatten' attribute. The default is 'true'. If you set it to 'false', the structure is a `Map` of `Map` objects.

Consider the following example:

```
public class Parent {
    private Child child;
    private String name;
    // setters and getters are omitted
}

public class Child {
    private String name;
    private List<String> nickNames;
    // setters and getters are omitted
}
```

The two classes in the preceding example are transformed to the following `Map`:

```
{name=George, child={name=Jenna, nickNames=[Bimbo, ...]}}
```

To configure these transformers, Spring Integration provides namespace support for Object-to-Map, as the following example shows:

```
<int:object-to-map-transformer input-channel="directInput" output-channel="output
"/>
```

You can also set the `flatten` attribute to false, as follows:

```
<int:object-to-map-transformer input-channel="directInput" output-channel="output"
flatten="false"/>
```

Spring Integration provides namespace support for Map-to-Object, as the following example shows:

```
<int:map-to-object-transformer input-channel="input"
                           output-channel="output"
                            type="org.something.Person"/>
```

Alterately, you could use a `ref` attribute and a prototype-scoped bean, as the following example shows:

```
<int:map-to-object-transformer input-channel="inputA"
                               output-channel="outputA"
                               ref="person"/>
<bean id="person" class="org.something.Person" scope="prototype"/>
```

The 'ref' and 'type' attributes are mutually exclusive. Also, if you use the 'ref' attribute, you must point to a 'prototype' scoped bean. Otherwise, a `BeanCreationException` is thrown.

Starting with version 5.0, you can supply the `ObjectToMapTransformer` with a customized `JsonObjectMapper` — for when you need special formats for dates or nulls for empty collections (and other uses). See JSON Transformers for more information about `JsonObjectMapper` implementations.

Stream Transformer

The `StreamTransformer` transforms `InputStream` payloads to a `byte[]`(or a `String` if a `charset` is provided).

The following example shows how to use the `stream-transformer` element in XML:

```
<int:stream-transformer input-channel="directInput" output-channel="output"/> <!--
byte[] -->

<int:stream-transformer id="withCharset" charset="UTF-8"
    input-channel="charsetChannel" output-channel="output"/> <!-- String -->
```

The following example shows how to use the `StreamTransformer` class and the `@Transformer` annotation to configure a stream transformer in Java:

```
@Bean
@Transformer(inputChannel = "stream", outputChannel = "data")
public StreamTransformer streamToBytes() {
    return new StreamTransformer(); // transforms to byte[]
}

@Bean
@Transformer(inputChannel = "stream", outputChannel = "data")
public StreamTransformer streamToString() {
    return new StreamTransformer("UTF-8"); // transforms to String
}
```

JSON Transformers

Spring Integration provides Object-to-JSON and JSON-to-Object transformers. The following pair of examples show how to declare them in XML:

```
<int:object-to-json-transformer input-channel="objectMapperInput"/>
```

```
<int:json-to-object-transformer input-channel="objectMapperInput"
    type="foo.MyDomainObject"/>
```

By default, the transformers in the preceding listing use a vanilla `JsonObjectMapper`. It is based on an implementation from the classpath. You can provide your own custom `JsonObjectMapper` implementation with appropriate options or based on a required library (such as GSON), as the following example shows:

```
<int:json-to-object-transformer input-channel="objectMapperInput"
    type="something.MyDomainObject" object-mapper="customObjectMapper"/>
```

Beginning with version 3.0, the `object-mapper` attribute references an instance of a new strategy interface: `JsonObjectMapper`. This abstraction lets multiple implementations of JSON mappers be used. Implementation that wraps Jackson 2 is provided, with the version being detected on the classpath. The class is `Jackson2JsonObjectMapper`, respectively.

The `BoonJsonObjectMapper` is deprecated in 5.2 since the library is out of support.

If you have requirements to use both Jackson and Boon in the same application, keep in mind that, before version 3.0, the JSON transformers used only Jackson 1.x. From 4.1 on, the framework selects Jackson 2 by default. Jackson 1.x is no longer supported by the framework internally. However, you can still use it within your code by including the necessary library. To avoid unexpected issues with JSON mapping features when you use annotations, you may need to apply annotations from both Jackson and Boon on domain classes, as the following example shows:

```
@com.fasterxml.jackson.annotation.JsonIgnoreProperties(ignoreUnknown=tr
ue)
public class Thing1 {

        @com.fasterxml.jackson.annotation.JsonProperty("thing1Thing2")
        public Object thing2;

}
```

Boon support has been deprecated since version 5.2.

You may wish to consider using a `FactoryBean` or a factory method to create the `JsonObjectMapper` with the required characteristics. The following example shows how to use such a factory:

```
public class ObjectMapperFactory {

    public static Jackson2JsonObjectMapper getMapper() {
        ObjectMapper mapper = new ObjectMapper();
        mapper.configure(JsonParser.Feature.ALLOW_COMMENTS, true);
        return new Jackson2JsonObjectMapper(mapper);
    }
}
```

The following example shows how to do the same thing in XML

```
<bean id="customObjectMapper" class="something.ObjectMapperFactory"
            factory-method="getMapper"/>
```

Beginning with version 2.2, the `object-to-json-transformer` sets the `content-type` header to `application/json`, by default, if the input message does not already have that header.

It you wish to set the `content-type` header to some other value or explicitly overwrite any existing header with some value (including `application/json`), use the `content-type` attribute. If you wish to suppress the setting of the header, set the `content-type` attribute to an empty string (`""`). Doing so results in a message with no `content-type` header, unless such a header was present on the input message.

Beginning with version 3.0, the `ObjectToJsonTransformer` adds headers, reflecting the source type, to the message. Similarly, the `JsonToObjectTransformer` can use those type headers when converting the JSON to an object. These headers are mapped in the AMQP adapters so that they are entirely compatible with the Spring-AMQP `JsonMessageConverter`.

This enables the following flows to work without any special configuration:

- `···→amqp-outbound-adapter---→`
- `---→amqp-inbound-adapter→json-to-object-transformer→···`

  Where the outbound adapter is configured with a `JsonMessageConverter` and the inbound adapter uses the default `SimpleMessageConverter`.

- `···→object-to-json-transformer→amqp-outbound-adapter---→`
- `---→amqp-inbound-adapter→···`

  Where the outbound adapter is configured with a `SimpleMessageConverter` and the inbound adapter uses the default `JsonMessageConverter`.

- `···→object-to-json-transformer→amqp-outbound-adapter---→`
- `---→amqp-inbound-adapter→json-to-object-transformer→`

  Where both adapters are configured with a `SimpleMessageConverter`.

When using the headers to determine the type, you should not provide a `class` attribute, because it takes precedence over the headers.

In addition to JSON Transformers, Spring Integration provides a built-in `#jsonPath` SpEL function for use in expressions. For more information see Spring Expression Language (SpEL).

Since version 3.0, Spring Integration also provides a built-in `#xpath` SpEL function for use in expressions. For more information see #xpath SpEL Function.

Beginning with version 4.0, the `ObjectToJsonTransformer` supports the `resultType` property, to specify the node JSON representation. The result node tree representation depends on the implementation of the provided `JsonObjectMapper`. By default, the `ObjectToJsonTransformer` uses a `Jackson2JsonObjectMapper` and delegates the conversion of the object to the node tree to the `ObjectMapper#valueToTree` method. The node JSON representation provides efficiency for using the `JsonPropertyAccessor` when the downstream message flow uses SpEL expressions with access to the properties of the JSON data. See Property Accessors for more information.

Beginning with version 5.1, the `resultType` can be configured as `BYTES` to produce a message with the `byte[]` payload for convenience when working with downstream handlers which operate with this data type.

Starting with version 5.2, the `JsonToObjectTransformer` can be configured with a `ResolvableType` to support generics during deserialization with the target JSON processor. Also this component now consults request message headers first for the presence of the `JsonHeaders.RESOLVABLE_TYPE` or `JsonHeaders.TYPE_ID` and falls back to the configured type otherwise. The `ObjectToJsonTransformer` now also populates a `JsonHeaders.RESOLVABLE_TYPE` header based on the request message payload for any possible downstream scenarios.

Starting with version 5.2.6, the `JsonToObjectTransformer` can be supplied with a `valueTypeExpression` to resolve a `ResolvableType` for the payload to convert from JSON at runtime against the request message. By default it consults `JsonHeaders` in the request message. If this expression returns `null` or `ResolvableType` building throws a `ClassNotFoundException`, the transformer falls back to the provided `targetType`. This logic is present as an expression because `JsonHeaders` may not have real class values, but rather some type ids which have to be mapped to target classes according some external registry.

Apache Avro Transformers

Version 5.2 added simple transformers to transform to/from Apache Avro.

They are unsophisticated in that there is no schema registry; the transformers simply use the schema embedded in the `SpecificRecord` implementation generated from the Avro schema.

Messages sent to the `SimpleToAvroTransformer` must have a payload that implements `SpecificRecord`; the transformer can handle multiple types. The `SimpleFromAvroTransformer` must be configured with a `SpecificRecord` class which is used as the default type to deserialize. You can also specify a SpEL expression to determine the type to deserialize using the `setTypeExpression` method. The default SpEL expression is `headers[avro_type]` (`AvroHeaders.TYPE`) which, by default, is populated by the `SimpleToAvroTransformer` with the fully qualified class name of the source class. If the expression returns `null`, the `defaultType` is used.

The `SimpleToAvroTransformer` also has a `setTypeExpression` method. This allows decoupling of the producer and consumer where the sender can set the header to some token representing the type and the consumer then maps that token to a type.

9.1.4. Configuring a Transformer with Annotations

You can add the `@Transformer` annotation to methods that expect either the `Message` type or the message payload type. The return value is handled in the exact same way as described earlier in

the section describing the `<transformer>` element. The following example shows how to use the `@Transformer` annotation to transform a `String` into an `Order`:

```
@Transformer
Order generateOrder(String productId) {
    return new Order(productId);
}
```

Transformer methods can also accept the `@Header` and `@Headers` annotations, as documented in `Annotation Support`. The following examples shows how to use the `@Header` annotation:

```
@Transformer
Order generateOrder(String productId, @Header("customerName") String customer) {
    return new Order(productId, customer);
}
```

See also Advising Endpoints Using Annotations.

9.1.5. Header Filter

Sometimes, your transformation use case might be as simple as removing a few headers. For such a use case, Spring Integration provides a header filter that lets you specify certain header names that should be removed from the output message (for example, removing headers for security reasons or a value that was needed only temporarily). Basically, the header filter is the opposite of the header enricher. The latter is discussed in Header Enricher. The following example defines a header filter:

```
<int:header-filter input-channel="inputChannel"
        output-channel="outputChannel" header-names="lastName, state"/>
```

As you can see, configuration of a header filter is quite simple. It is a typical endpoint with input and output channels and a `header-names` attribute. That attribute accepts the names of the headers (delimited by commas if there are multiple) that need to be removed. So, in the preceding example, the headers named 'lastName' and 'state' are not present on the outbound message.

9.1.6. Codec-Based Transformers

See Codec.

9.2. Content Enricher

At times, you may have a requirement to enhance a request with more information than was provided by the target system. The data enricher pattern describes various scenarios as well as the component (Enricher) that lets you address such requirements.

The Spring Integration `Core` module includes two enrichers:

- Header Enricher
- Payload Enricher

It also includes three adapter-specific header enrichers:

- XPath Header Enricher (XML Module)
- Mail Header Enricher (Mail Module)
- XMPP Header Enricher (XMPP Module)

See the adapter-specific sections of this reference manual to learn more about those adapters.

For more information regarding expressions support, see Spring Expression Language (SpEL).

9.2.1. Header Enricher

If you need do nothing more than add headers to a message and the headers are not dynamically determined by the message content, referencing a custom implementation of a transformer may be overkill. For that reason, Spring Integration provides support for the header enricher pattern. It is exposed through the `<header-enricher>` element. The following example shows how to use it:

```
<int:header-enricher input-channel="in" output-channel="out">
    <int:header name="foo" value="123"/>
    <int:header name="bar" ref="someBean"/>
</int:header-enricher>
```

The header enricher also provides helpful sub-elements to set well known header names, as the following example shows:

```
<int:header-enricher input-channel="in" output-channel="out">
    <int:error-channel ref="applicationErrorChannel"/>
    <int:reply-channel ref="quoteReplyChannel"/>
    <int:correlation-id value="123"/>
    <int:priority value="HIGHEST"/>
    <routing-slip value="channel1; routingSlipRoutingStrategy;
request.headers[myRoutingSlipChannel]"/>
    <int:header name="bar" ref="someBean"/>
</int:header-enricher>
```

The preceding configuration shows that, for well known headers (such as `errorChannel`, `correlationId`, `priority`, `replyChannel`, `routing-slip`, and others), instead of using generic `<header>` sub-elements where you would have to provide both header 'name' and 'value', you can use convenient sub-elements to set those values directly.

Starting with version 4.1, the header enricher provides a `routing-slip` sub-element. See Routing Slip for more information.

POJO Support

Often, a header value cannot be defined statically and has to be determined dynamically based on some content in the message. That is why the header enricher lets you also specify a bean reference by using the `ref` and `method` attributes. The specified method calculates the header value. Consider the following configuration and a bean with a method that modifies a `String`:

```
<int:header-enricher input-channel="in" output-channel="out">
    <int:header name="something" method="computeValue" ref="myBean"/>
</int:header-enricher>

<bean id="myBean" class="thing1.thing2.MyBean"/>
```

```
public class MyBean {

    public String computeValue(String payload){
        return payload.toUpperCase() + "_US";
    }
}
```

You can also configure your POJO as an inner bean, as the following example shows:

```
<int:header-enricher  input-channel="inputChannel" output-channel="outputChannel">
    <int:header name="some_header">
        <bean class="org.MyEnricher"/>
    </int:header>
</int:header-enricher>
```

You can similarly point to a Groovy script, as the following example shows:

```
<int:header-enricher  input-channel="inputChannel" output-channel="outputChannel">
    <int:header name="some_header">
        <int-groovy:script location="org/SampleGroovyHeaderEnricher.groovy"/>
    </int:header>
</int:header-enricher>
```

SpEL Support

In Spring Integration 2.0, we introduced the convenience of the Spring Expression Language (SpEL) to help configure many different components. The header enricher is one of them. Look again at the POJO example shown earlier. You can see that the computation logic to determine the header value is pretty simple. A natural question would be: "Is there an even simpler way to accomplish this?". That is where SpEL shows its true power. Consider the following example:

```
<int:header-enricher input-channel="in" output-channel="out">
    <int:header name="foo" expression="payload.toUpperCase() + '_US'"/>
</int:header-enricher>
```

By using SpEL for such simple cases, you no longer have to provide a separate class and configure it in the application context. All you need do is configured the `expression` attribute with a valid SpEL expression. The 'payload' and 'headers' variables are bound to the SpEL evaluation context, giving you full access to the incoming message.

Configuring a Header Enricher with Java Configuration

The following two examples show how to use Java Configuration for header enrichers:

```
@Bean
@Transformer(inputChannel = "enrichHeadersChannel", outputChannel = "emailChannel
")
public HeaderEnricher enrichHeaders() {
    Map<String, ? extends HeaderValueMessageProcessor<?>> headersToAdd =
            Collections.singletonMap("emailUrl",
                    new StaticHeaderValueMessageProcessor<>(this.imapUrl));
    HeaderEnricher enricher = new HeaderEnricher(headersToAdd);
    return enricher;
}

@Bean
@Transformer(inputChannel="enrichHeadersChannel", outputChannel="emailChannel")
public HeaderEnricher enrichHeaders() {
    Map<String, HeaderValueMessageProcessor<?>> headersToAdd = new HashMap<>();
    headersToAdd.put("emailUrl", new StaticHeaderValueMessageProcessor<String>
(this.imapUrl));
    Expression expression = new SpelExpressionParser().parseExpression(
"payload.from[0].toString()");
    headersToAdd.put("from",
                new ExpressionEvaluatingHeaderValueMessageProcessor<>(expression,
String.class));
    HeaderEnricher enricher = new HeaderEnricher(headersToAdd);
    return enricher;
}
```

The first example adds a single literal header. The second example adds two headers, a literal header and one based on a SpEL expression.

Configuring a Header Enricher with the Java DSL

The following example shows Java DSL Configuration for a header enricher:

```
@Bean
public IntegrationFlow enrichHeadersInFlow() {
    return f -> f
                ...
                .enrichHeaders(h -> h.header("emailUrl", this.emailUrl)
                                     .headerExpression("from",
"payload.from[0].toString()"))
                .handle(...);
}
```

Header Channel Registry

Starting with Spring Integration 3.0, a new sub-element `<int:header-channels-to-string/>` is available. It has no attributes. This new sub-element converts existing `replyChannel` and `errorChannel` headers (when they are a `MessageChannel`) to a `String` and stores the channels in a registry for later resolution, when it is time to send a reply or handle an error. This is useful for cases where the headers might be lost — for example, when serializing a message into a message store or when transporting the message over JMS. If the header does not already exist or it is not a `MessageChannel`, no changes are made.

Using this functionality requires the presence of a `HeaderChannelRegistry` bean. By default, the framework creates a `DefaultHeaderChannelRegistry` with the default expiry (60 seconds). Channels are removed from the registry after this time. To change this behavior, define a bean with an `id` of `integrationHeaderChannelRegistry` and configure the required default delay by using a constructor argument (in milliseconds).

Since version 4.1, you can set a property called `removeOnGet` to `true` on the `<bean/>` definition, and the mapping entry is removed immediately on first use. This might be useful in a high-volume environment and when the channel is only used once, rather than waiting for the reaper to remove it.

The `HeaderChannelRegistry` has a `size()` method to determine the current size of the registry. The `runReaper()` method cancels the current scheduled task and runs the reaper immediately. The task is then scheduled to run again based on the current delay. These methods can be invoked directly by getting a reference to the registry, or you can send a message with, for example, the following content to a control bus:

```
"@integrationHeaderChannelRegistry.runReaper()"
```

This sub-element is a convenience, and is the equivalent of specifying the following configuration:

```
<int:reply-channel
    expression=
"@integrationHeaderChannelRegistry.channelToChannelName(headers.replyChannel)"
    overwrite="true" />
<int:error-channel
    expression=
"@integrationHeaderChannelRegistry.channelToChannelName(headers.errorChannel)"
    overwrite="true" />
```

Starting with version 4.1, you can now override the registry's configured reaper delay so that the the channel mapping is retained for at least the specified time, regardless of the reaper delay. The following example shows how to do so:

```
<int:header-enricher input-channel="inputTtl" output-channel="next">
    <int:header-channels-to-string time-to-live-expression="120000" />
</int:header-enricher>

<int:header-enricher input-channel="inputCustomTtl" output-channel="next">
    <int:header-channels-to-string
        time-to-live-expression="headers['channelTTL'] ?: 120000" />
</int:header-enricher>
```

In the first case, the time to live for every header channel mapping will be two minutes. In the second case, the time to live is specified in the message header and uses an Elvis operator to use two minutes if there is no header.

9.2.2. Payload Enricher

In certain situations, the header enricher, as discussed earlier, may not be sufficient and payloads themselves may have to be enriched with additional information. For example, order messages that enter the Spring Integration messaging system have to look up the order's customer based on the provided customer number and then enrich the original payload with that information.

Spring Integration 2.1 introduced the payload enricher. The payload enricher defines an endpoint that passes a `Message` to the exposed request channel and then expects a reply message. The reply message then becomes the root object for evaluation of expressions to enrich the target payload.

The payload enricher provides full XML namespace support through the `enricher` element. In order to send request messages, the payload enricher has a `request-channel` attribute that lets you dispatch messages to a request channel.

Basically, by defining the request channel, the payload enricher acts as a gateway, waiting for the message sent to the request channel to return. The enricher then augments the message's payload with the data provided by the reply message.

When sending messages to the request channel, you also have the option to send only a subset of the original payload by using the `request-payload-expression` attribute.

The enriching of payloads is configured through SpEL expressions, providing a maximum degree of flexibility. Therefore, you can not only enrich payloads with direct values from the reply channel's `Message`, but you can use SpEL expressions to extract a subset from that message or to apply additional inline transformations, letting you further manipulate the data.

If you need only to enrich payloads with static values, you need not provide the `request-channel` attribute.

Enrichers are a variant of transformers. In many cases, you could use a payload enricher or a generic transformer implementation to add additional data to your message payloads. You should familiarize yourself with all transformation-capable components that are provided by Spring Integration and carefully select the implementation that semantically fits your business case best.

Configuration

The following example shows all available configuration options for the payload enricher:

```
<int:enricher request-channel=""                                    ①
              auto-startup="true"                                   ②
              id=""                                                 ③
              order=""                                              ④
              output-channel=""                                     ⑤
              request-payload-expression=""                         ⑥
              reply-channel=""                                      ⑦
              error-channel=""                                      ⑧
              send-timeout=""                                       ⑨
              should-clone-payload="false">                         ⑩
    <int:poller></int:poller>                                       ⑪
    <int:property name="" expression="" null-result-expression="'Could not determine
the name'"/>   ⑫
    <int:property name="" value="23" type="java.lang.Integer" null-result-expression=
"'0'"/>
    <int:header name="" expression="" null-result-expression=""/>   ⑬
    <int:header name="" value="" overwrite="" type="" null-result-expression=""/>
</int:enricher>
```

① Channel to which a message is sent to get the data to use for enrichment. Optional.

② Lifecycle attribute signaling whether this component should be started during the application context startup. Defaults to true. Optional.

③ ID of the underlying bean definition, which is either an `EventDrivenConsumer` or a `PollingConsumer`. Optional.

④ Specifies the order for invocation when this endpoint is connected as a subscriber to a channel. This is particularly relevant when that channel is using a “failover” dispatching strategy. It has no effect when this endpoint is itself a polling consumer for a channel with a queue. Optional.

⑤ Identifies the message channel where a message is sent after it is being processed by this endpoint. Optional.

⑥ By default, the original message’s payload is used as payload that is sent to the `request-channel`. By specifying a SpEL expression as the value for the `request-payload-expression` attribute, you can use a subset of the original payload, a header value, or any other resolvable SpEL expression as the basis for the payload that is sent to the request-channel. For the expression evaluation, the full message is available as the 'root object'. For instance, the following SpEL expressions (among others) are possible:

- `payload.something`

- `headers.something`
- `new java.util.Date()`
- `'thing1' + 'thing2'`

⑦ Channel where a reply message is expected. This is optional. Typically, the auto-generated temporary reply channel suffices. Optional.

⑧ The channel to which an `ErrorMessage` is sent if an `Exception` occurs downstream of the `request-channel`. This enables you to return an alternative object to use for enrichment. If it is not set, an `Exception` is thrown to the caller. Optional.

⑨ Maximum amount of time in milliseconds to wait when sending a message to the channel, if the channel might block. For example, a queue channel can block until space is available, if its maximum capacity has been reached. Internally, the send timeout is set on the `MessagingTemplate` and ultimately applied when invoking the send operation on the `MessageChannel`. By default, the send timeout is set to '-1', which can cause the send operation on the `MessageChannel`, depending on the implementation, to block indefinitely. Optional.

⑩ Boolean value indicating whether any payload that implements `Cloneable` should be cloned prior to sending the message to the request channel for acquiring the enriching data. The cloned version would be used as the target payload for the ultimate reply. The default is `false`. Optional.

⑪ Lets you configure a message poller if this endpoint is a polling consumer. Optional.

⑫ Each `property` sub-element provides the name of a property (through the mandatory `name` attribute). That property should be settable on the target payload instance. Exactly one of the `value` or `expression` attributes must be provided as well — the former for a literal value to set and the latter for a SpEL expression to be evaluated. The root object of the evaluation context is the message that was returned from the flow initiated by this enricher — the input message if there is no request channel or the application context (using the '@<beanName>.<beanProperty>' SpEL syntax). Starting with version 4.0, when specifying a `value` attribute, you can also specify an optional `type` attribute. When the destination is a typed setter method, the framework coerces the value appropriately (as long as a `PropertyEditor`) exists to handle the conversion. If, however, the target payload is a `Map`, the entry is populated with the value without conversion. The `type` attribute lets you, for example, convert a `String` containing a number to an `Integer` value in the target payload. Starting with version 4.1, you can also specify an optional `null-result-expression` attribute. When the `enricher` returns null, it is evaluated, and the output of the evaluation is returned instead.

⑬ Each `header` sub-element provides the name of a message header (through the mandatory `name` attribute). Exactly one of the `value` or `expression` attributes must also be provided — the former for a literal value to set and the latter for a SpEL expression to be evaluated. The root object of the evaluation context is the message that was returned from the flow initiated by this enricher — the input message if there is no request channel or the application context (using the '@<beanName>.<beanProperty>' SpEL syntax). Note that, similarly to the `<header-enricher>`, the `<enricher>` element's `header` element has `type` and `overwrite` attributes. However, a key difference is that, with the `<enricher>`, the `overwrite` attribute is `true` by default, to be consistent with the `<enricher>` element's `<property>` sub-element. Starting with version 4.1, you can also specify an optional `null-result-expression` attribute. When the `enricher` returns null, it is evaluated, and the output of the evaluation is returned instead.

Examples

This section contains several examples of using a payload enricher in various situations.

The code samples shown here are part of the Spring Integration Samples project. See Spring Integration Samples.

In the following example, a `User` object is passed as the payload of the `Message`:

```
<int:enricher id="findUserEnricher"
              input-channel="findUserEnricherChannel"
              request-channel="findUserServiceChannel">
    <int:property name="email"    expression="payload.email"/>
    <int:property name="password" expression="payload.password"/>
</int:enricher>
```

The `User` has several properties, but only the `username` is set initially. The enricher's `request-channel` attribute is configured to pass the `User` to the `findUserServiceChannel`.

Through the implicitly set `reply-channel`, a `User` object is returned and, by using the `property` sub-element, properties from the reply are extracted and used to enrich the original payload.

How Do I Pass Only a Subset of Data to the Request Channel?

When using a `request-payload-expression` attribute, a single property of the payload instead of the full message can be passed on to the request channel. In the following example, the username property is passed on to the request channel:

```
<int:enricher id="findUserByUsernameEnricher"
              input-channel="findUserByUsernameEnricherChannel"
              request-channel="findUserByUsernameServiceChannel"
              request-payload-expression="payload.username">
    <int:property name="email"    expression="payload.email"/>
    <int:property name="password" expression="payload.password"/>
</int:enricher>
```

Keep in mind that, although only the username is passed, the resulting message to the request channel contains the full set of `MessageHeaders`.

How Can I Enrich Payloads that Consist of Collection Data?

In the following example, instead of a `User` object, a `Map` is passed in:

```
<int:enricher id="findUserWithMapEnricher"
              input-channel="findUserWithMapEnricherChannel"
              request-channel="findUserByUsernameServiceChannel"
              request-payload-expression="payload.username">
    <int:property name="user" expression="payload"/>
</int:enricher>
```

The `Map` contains the username under the `username` map key. Only the `username` is passed on to the request channel. The reply contains a full `User` object, which is ultimately added to the `Map` under the `user` key.

How Can I Enrich Payloads with Static Information without Using a Request Channel?

The following example does not use a request channel at all but solely enriches the message's payload with static values:

```
<int:enricher id="userEnricher"
              input-channel="input">
    <int:property name="user.updateDate" expression="new java.util.Date()"/>
    <int:property name="user.firstName" value="William"/>
    <int:property name="user.lastName"  value="Shakespeare"/>
    <int:property name="user.age"       value="42"/>
</int:enricher>
```

Note that the word, 'static', is used loosely here. You can still use SpEL expressions for setting those values.

9.3. Claim Check

In earlier sections, we covered several content enricher components that can help you deal with situations where a message is missing a piece of data. We also discussed content filtering, which lets you remove data items from a message. However, there are times when we want to hide data temporarily. For example, in a distributed system, we may receive a message with a very large payload. Some intermittent message processing steps may not need access to this payload and some may only need to access certain headers, so carrying the large message payload through each processing step may cause performance degradation, may produce a security risk, and may make debugging more difficult.

The store in library (or claim check) pattern describes a mechanism that lets you store data in a well known place while maintaining only a pointer (a claim check) to where that data is located. You can pass that pointer around as the payload of a new message, thereby letting any component within the message flow get the actual data as soon as it needs it. This approach is very similar to the certified mail process, where you get a claim check in your mailbox and then have to go to the post office to claim your actual package. It is also the same idea as baggage claim after a flight or in

a hotel.

Spring Integration provides two types of claim check transformers:

- Incoming Claim Check Transformer
- Outgoing Claim Check Transformer

Convenient namespace-based mechanisms are available to configure them.

9.3.1. Incoming Claim Check Transformer

An incoming claim check transformer transforms an incoming message by storing it in the message store identified by its `message-store` attribute. The following example defines an incoming claim check transformer:

```
<int:claim-check-in id="checkin"
        input-channel="checkinChannel"
        message-store="testMessageStore"
        output-channel="output"/>
```

In the preceding configuration, the message that is received on the `input-channel` is persisted to the message store identified with the `message-store` attribute and indexed with a generated ID. That ID is the claim check for that message. The claim check also becomes the payload of the new (transformed) message that is sent to the `output-channel`.

Now, assume that at some point you do need access to the actual message. You can access the message store manually and get the contents of the message, or you can use the same approach (creating a transformer) except that now you transform the Claim Check to the actual message by using an outgoing claim check transformer.

The following listing provides an overview of all available parameters of an incoming claim check transformer:

```
<int:claim-check-in auto-startup="true"  ①
                    id=""                ②
                    input-channel=""     ③
                    message-store="messageStore"  ④
                    order=""             ⑤
                    output-channel=""    ⑥
                    send-timeout="">     ⑦
    <int:poller></int:poller>            ⑧
</int:claim-check-in>
```

① Lifecycle attribute signaling whether this component should be started during application context startup. It defaults to `true`. This attribute is not available inside a `Chain` element. Optional.

② ID identifying the underlying bean definition (`MessageTransformingHandler`). This attribute is not available inside a `Chain` element. Optional.

③ The receiving message channel of this endpoint. This attribute is not available inside a `Chain` element. Optional.

④ Reference to the `MessageStore` to be used by this claim check transformer. If not specified, the default reference is to a bean named `messageStore`. Optional.

⑤ Specifies the order for invocation when this endpoint is connected as a subscriber to a channel. This is particularly relevant when that channel uses a `failover` dispatching strategy. It has no effect when this endpoint is itself a polling consumer for a channel with a queue. This attribute is not available inside a `Chain` element. Optional.

⑥ Identifies the message channel where the message is sent after being processed by this endpoint. This attribute is not available inside a `Chain` element. Optional.

⑦ Specifies the maximum amount of time (in milliseconds) to wait when sending a reply message to the output channel. Defaults to `-1` — blocking indefinitely. This attribute is not available inside a `Chain` element. Optional.

⑧ Defines a poller. This element is not available inside a `Chain` element. Optional.

9.3.2. Outgoing Claim Check Transformer

An outgoing claim check transformer lets you transform a message with a claim check payload into a message with the original content as its payload.

```
<int:claim-check-out id="checkout"
        input-channel="checkoutChannel"
        message-store="testMessageStore"
        output-channel="output"/>
```

In the preceding configuration, the message received on the `input-channel` should have a claim

check as its payload. The outgoing claim check transformer transforms it into a message with the original payload by querying the message store for a message identified by the provided claim check. It then sends the newly checked-out message to the `output-channel`.

The following listing provides an overview of all available parameters of an outgoing claim check transformer:

```
<int:claim-check-out auto-startup="true"  ①
                     id=""                ②
                     input-channel=""     ③
                     message-store="messageStore"  ④
                     order=""             ⑤
                     output-channel=""    ⑥
                     remove-message="false"  ⑦
                     send-timeout="">     ⑧
    <int:poller></int:poller>             ⑨
</int:claim-check-out>
```

① Lifecycle attribute signaling whether this component should be started during application context startup. It defaults to `true`. This attribute is not available inside a `Chain` element. Optional.

② ID identifying the underlying bean definition (`MessageTransformingHandler`). This attribute is not available inside a `Chain` element. Optional.

③ The receiving message channel of this endpoint. This attribute is not available inside a `Chain` element. Optional.

④ Reference to the `MessageStore` to be used by this claim check transformer. If not specified, the default reference is to a bean named `messageStore`. Optional.

⑤ Specifies the order for invocation when this endpoint is connected as a subscriber to a channel. This is particularly relevant when that channel is using a `failover` dispatching strategy. It has no effect when this endpoint is itself a polling consumer for a channel with a queue. This attribute is not available inside a `Chain` element. Optional.

⑥ Identifies the message channel where the message is sent after being processed by this endpoint. This attribute is not available inside a `Chain` element. Optional.

⑦ If set to `true`, the message is removed from the `MessageStore` by this transformer. This setting is useful when Message can be “claimed” only once. It defaults to `false`. Optional.

⑧ Specifies the maximum amount of time (in milliseconds) to wait when sending a reply message to the output channel. It defaults to `-1` — blocking indefinitely. This attribute is not available inside a `Chain` element. Optional.

⑨ Defines a poller. This element is not available inside a `Chain` element. Optional.

9.3.3. Claim Once

Sometimes, a particular message must be claimed only once. As an analogy, consider process of handling airplane luggage. You checking in your luggage on departure and claiming it on arrival.

Once the luggage has been claimed, it can not be claimed again without first checking it back in. To accommodate such cases, we introduced a `remove-message` boolean attribute on the `claim-check-out` transformer. This attribute is set to `false` by default. However, if set to `true`, the claimed message is removed from the `MessageStore` so that it cannot be claimed again.

This feature has an impact in terms of storage space, especially in the case of the in-memory `Map` -based `SimpleMessageStore`, where failing to remove messages could ultimately lead to an `OutOfMemoryException`. Therefore, if you do not expect multiple claims to be made, we recommend that you set the `remove-message` attribute's value to `true`. The following example show how to use the `remove-message` attribute:

```
<int:claim-check-out id="checkout"
        input-channel="checkoutChannel"
        message-store="testMessageStore"
        output-channel="output"
        remove-message="true"/>
```

9.3.4. A Word on Message Store

Although we rarely care about the details of the claim checks (as long as they work), you should know that the current implementation of the actual claim check (the pointer) in Spring Integration uses a UUID to ensure uniqueness.

`org.springframework.integration.store.MessageStore` is a strategy interface for storing and retrieving messages. Spring Integration provides two convenient implementations of it:

- `SimpleMessageStore`: An in-memory, `Map`-based implementation (the default, good for testing)
- `JdbcMessageStore`: An implementation that uses a relational database over JDBC

9.4. Codec

Version 4.2 of Spring Integration introduced the `Codec` abstraction. Codecs encode and decode objects to and from `byte[]`. They offer an alternative to Java serialization. One advantage is that, typically, objects need not implement `Serializable`. We provide one implementation that uses Kryo for serialization, but you can provide your own implementation for use in any of the following components:

- `EncodingPayloadTransformer`
- `DecodingTransformer`
- `CodecMessageConverter`

9.4.1. `EncodingPayloadTransformer`

This transformer encodes the payload to a `byte[]` by using the codec. It does not affect message headers.

See the Javadoc for more information.

9.4.2. DecodingTransformer

This transformer decodes a `byte[]` by using the codec. It needs to be configured with the `Class` to which the object should be decoded (or an expression that resolves to a `Class`). If the resulting object is a `Message<?>`, inbound headers are not retained.

See the Javadoc for more information.

9.4.3. CodecMessageConverter

Certain endpoints (such as TCP and Redis) have no concept of message headers. They support the use of a `MessageConverter`, and the `CodecMessageConverter` can be used to convert a message to or from a `byte[]` for transmission.

See the Javadoc for more information.

9.4.4. Kryo

Currently, this is the only implementation of `Codec`, and it provides two kinds of `Codec`:

- `PojoCodec`: Used in the transformers
- `MessageCodec`: Used in the `CodecMessageConverter`

The framework provides several custom serializers:

- `FileSerializer`
- `MessageHeadersSerializer`
- `MutableMessageHeadersSerializer`

The first can be used with the `PojoCodec` by initializing it with the `FileKryoRegistrar`. The second and third are used with the `MessageCodec`, which is initialized with the `MessageKryoRegistrar`.

Customizing Kryo

By default, Kryo delegates unknown Java types to its `FieldSerializer`. Kryo also registers default serializers for each primitive type, along with `String`, `Collection`, and `Map`. `FieldSerializer` uses reflection to navigate the object graph. A more efficient approach is to implement a custom serializer that is aware of the object's structure and can directly serialize selected primitive fields. The following example shows such a serializer:

```
public class AddressSerializer extends Serializer<Address> {

    @Override
    public void write(Kryo kryo, Output output, Address address) {
        output.writeString(address.getStreet());
        output.writeString(address.getCity());
        output.writeString(address.getCountry());
    }

    @Override
    public Address read(Kryo kryo, Input input, Class<Address> type) {
        return new Address(input.readString(), input.readString(), input
.readString());
    }
}
```

The `Serializer` interface exposes `Kryo`, `Input`, and `Output`, which provide complete control over which fields are included and other internal settings, as described in the Kryo documentation.

When registering your custom serializer, you need a registration ID. The registration IDs are arbitrary. However, in our case, the IDs must be explicitly defined, because each Kryo instance across the distributed application must use the same IDs. Kryo recommends small positive integers and reserves a few ids (value < 10). Spring Integration currently defaults to using 40, 41, and 42 (for the file and message header serializers mentioned earlier). We recommend you start at 60, to allow for expansion in the framework. You can override these framework defaults by configuring the registrars mentioned earlier.

Using a Custom Kryo Serializer

If you need custom serialization, see the Kryo documentation, because you need to use the native API to do the customization. For an example, see the `MessageCodec` implementation.

Implementing KryoSerializable

If you have write access to the domain object source code, you can implement `KryoSerializable` as described here. In this case, the class provides the serialization methods itself and no further configuration is required. However benchmarks have shown this is not quite as efficient as registering a custom serializer explicitly. The following example shows a custom Kryo serializer:

```
public class Address implements KryoSerializable {
    ...

    @Override
    public void write(Kryo kryo, Output output) {
        output.writeString(this.street);
        output.writeString(this.city);
        output.writeString(this.country);
    }

    @Override
    public void read(Kryo kryo, Input input) {
        this.street = input.readString();
        this.city = input.readString();
        this.country = input.readString();
    }
}
```

You can also use this technique to wrap a serialization library other than Kryo.

Using the `@DefaultSerializer` Annotation

Kryo also provides a `@DefaultSerializer` annotation, as described here.

```
@DefaultSerializer(SomeClassSerializer.class)
public class SomeClass {
       // ...
}
```

If you have write access to the domain object, this may be a simpler way to specify a custom serializer. Note that this does not register the class with an ID, which may make the technique unhelpful for certain situations.

Chapter 10. Messaging Endpoints

10.1. Message Endpoints

The first part of this chapter covers some background theory and reveals quite a bit about the underlying API that drives Spring Integration's various messaging components. This information can be helpful if you want to really understand what goes on behind the scenes. However, if you want to get up and running with the simplified namespace-based configuration of the various elements, feel free to skip ahead to Endpoint Namespace Support for now.

As mentioned in the overview, message endpoints are responsible for connecting the various messaging components to channels. Over the next several chapters, we cover a number of different components that consume messages. Some of these are also capable of sending reply messages. Sending messages is quite straightforward. As shown earlier in Message Channels, you can send a message to a message channel. However, receiving is a bit more complicated. The main reason is that there are two types of consumers: polling consumers and event-driven consumers.

Of the two, event-driven consumers are much simpler. Without any need to manage and schedule a separate poller thread, they are essentially listeners with a callback method. When connecting to one of Spring Integration's subscribable message channels, this simple option works great. However, when connecting to a buffering, pollable message channel, some component has to schedule and manage the polling threads. Spring Integration provides two different endpoint implementations to accommodate these two types of consumers. Therefore, the consumers themselves need only implement the callback interface. When polling is required, the endpoint acts as a container for the consumer instance. The benefit is similar to that of using a container for hosting message-driven beans, but, since these consumers are Spring-managed objects running within an `ApplicationContext`, it more closely resembles Spring's own `MessageListener` containers.

10.1.1. Message Handler

Spring Integration's `MessageHandler` interface is implemented by many of the components within the framework. In other words, this is not part of the public API, and you would not typically implement `MessageHandler` directly. Nevertheless, it is used by a message consumer for actually handling the consumed messages, so being aware of this strategy interface does help in terms of understanding the overall role of a consumer. The interface is defined as follows:

```
public interface MessageHandler {

    void handleMessage(Message<?> message);

}
```

Despite its simplicity, this interface provides the foundation for most of the components (routers, transformers, splitters, aggregators, service activators, and others) covered in the following chapters. Those components each perform very different functionality with the messages they

handle, but the requirements for actually receiving a message are the same, and the choice between polling and event-driven behavior is also the same. Spring Integration provides two endpoint implementations that host these callback-based handlers and let them be connected to message channels.

10.1.2. Event-driven Consumer

Because it is the simpler of the two, we cover the event-driven consumer endpoint first. You may recall that the `SubscribableChannel` interface provides a `subscribe()` method and that the method accepts a `MessageHandler` parameter (as shown in `SubscribableChannel`). The following listing shows the definition of the `subscribe` method:

```
subscribableChannel.subscribe(messageHandler);
```

Since a handler that is subscribed to a channel does not have to actively poll that channel, this is an event-driven consumer, and the implementation provided by Spring Integration accepts a `SubscribableChannel` and a `MessageHandler`, as the following example shows:

```
SubscribableChannel channel = context.getBean("subscribableChannel",
SubscribableChannel.class);

EventDrivenConsumer consumer = new EventDrivenConsumer(channel, exampleHandler);
```

10.1.3. Polling Consumer

Spring Integration also provides a `PollingConsumer`, and it can be instantiated in the same way except that the channel must implement `PollableChannel`, as the following example shows:

```
PollableChannel channel = context.getBean("pollableChannel", PollableChannel.
class);

PollingConsumer consumer = new PollingConsumer(channel, exampleHandler);
```

For more information regarding polling consumers, see Poller and Channel Adapter.

There are many other configuration options for the polling consumer. For example, the trigger is a required property. The following example shows how to set the trigger:

```
PollingConsumer consumer = new PollingConsumer(channel, handler);

consumer.setTrigger(new IntervalTrigger(30, TimeUnit.SECONDS));
```

Spring Integration currently provides two implementations of the `Trigger` interface: `IntervalTrigger` and `CronTrigger`. The `IntervalTrigger` is typically defined with a simple interval (in milliseconds) but also supports an `initialDelay` property and a boolean `fixedRate` property (the default is `false` — that is, no fixed delay). The following example sets both properties:

```
IntervalTrigger trigger = new IntervalTrigger(1000);
trigger.setInitialDelay(5000);
trigger.setFixedRate(true);
```

The result of the three settings in the preceding example is a trigger that waits five seconds and then triggers every second.

The `CronTrigger` requires a valid cron expression. See the Javadoc for details. The following example sets a new `CronTrigger`:

```
CronTrigger trigger = new CronTrigger("*/10 * * * * MON-FRI");
```

The result of the trigger defined in the previous example is a trigger that triggers every ten seconds, Monday through Friday.

In addition to the trigger, you can specify two other polling-related configuration properties: `maxMessagesPerPoll` and `receiveTimeout`. The following example shows how to set these two properties:

```
PollingConsumer consumer = new PollingConsumer(channel, handler);

consumer.setMaxMessagesPerPoll(10);
consumer.setReceiveTimeout(5000);
```

The `maxMessagesPerPoll` property specifies the maximum number of messages to receive within a given poll operation. This means that the poller continues calling receive() without waiting, until either `null` is returned or the maximum value is reached. For example, if a poller has a ten-second interval trigger and a `maxMessagesPerPoll` setting of `25`, and it is polling a channel that has 100 messages in its queue, all 100 messages can be retrieved within 40 seconds. It grabs 25, waits ten seconds, grabs the next 25, and so on.

The `receiveTimeout` property specifies the amount of time the poller should wait if no messages are available when it invokes the receive operation. For example, consider two options that seem similar on the surface but are actually quite different: The first has an interval trigger of 5 seconds and a receive timeout of 50 milliseconds, while the second has an interval trigger of 50 milliseconds and a receive timeout of 5 seconds. The first one may receive a message up to 4950 milliseconds later than it arrived on the channel (if that message arrived immediately after one of its poll calls returned). On the other hand, the second configuration never misses a message by more than 50 milliseconds. The difference is that the second option requires a thread to wait. However, as a result, it can respond much more quickly to arriving messages. This technique, known as "long polling", can be used to emulate event-driven behavior on a polled source.

A polling consumer can also delegate to a Spring `TaskExecutor`, as the following example shows:

```
PollingConsumer consumer = new PollingConsumer(channel, handler);

TaskExecutor taskExecutor = context.getBean("exampleExecutor", TaskExecutor.class
);
consumer.setTaskExecutor(taskExecutor);
```

Furthermore, a `PollingConsumer` has a property called `adviceChain`. This property lets you to specify a `List` of AOP advices for handling additional cross cutting concerns including transactions. These advices are applied around the `doPoll()` method. For more in-depth information, see the sections on AOP advice chains and transaction support under Endpoint Namespace Support.

The earlier examples show dependency lookups. However, keep in mind that these consumers are most often configured as Spring bean definitions. In fact, Spring Integration also provides a `FactoryBean` called `ConsumerEndpointFactoryBean` that creates the appropriate consumer type based on the type of channel. Also, Spring Integration has full XML namespace support to even further hide those details. The namespace-based configuration is in this guide featured as each component type is introduced.

Many of the `MessageHandler` implementations can generate reply messages. As mentioned earlier, sending messages is trivial when compared to receiving messages. Nevertheless, when and how many reply messages are sent depends on the handler type. For example, an aggregator waits for a number of messages to arrive and is often configured as a downstream consumer for a splitter, which can generate multiple replies for each message it handles. When using the namespace configuration, you do not strictly need to know all of the details. However, it still might be worth knowing that several of these components share a common base class, the `AbstractReplyProducingMessageHandler`, and that it provides a `setOutputChannel(..)` method.

10.1.4. Endpoint Namespace Support

Throughout this reference manual, you can find specific configuration examples for endpoint elements, such as router, transformer, service-activator, and so on. Most of these support an `input-`

`channel` attribute and many support an `output-channel` attribute. After being parsed, these endpoint elements produce an instance of either the `PollingConsumer` or the `EventDrivenConsumer`, depending on the type of the `input-channel` that is referenced: `PollableChannel` or `SubscribableChannel`, respectively. When the channel is pollable, the polling behavior is based on the endpoint element's `poller` sub-element and its attributes.

The following listing lists all of the available configuration options for a `poller`:

```
<int:poller cron=""                                    ①
            default="false"                            ②
            error-channel=""                           ③
            fixed-delay=""                             ④
            fixed-rate=""                              ⑤
            id=""                                      ⑥
            max-messages-per-poll=""                   ⑦
            receive-timeout=""                         ⑧
            ref=""                                     ⑨
            task-executor=""                           ⑩
            time-unit="MILLISECONDS"                   ⑪
            trigger="">                                ⑫
            <int:advice-chain />                       ⑬
            <int:transactional />                      ⑭
</int:poller>
```

① Provides the ability to configure pollers by using Cron expressions. The underlying implementation uses an `org.springframework.scheduling.support.CronTrigger`. If this attribute is set, none of the following attributes must be specified: `fixed-delay`, `trigger`, `fixed-rate`, and `ref`.

② By setting this attribute to `true`, you can define exactly one global default poller. An exception is raised if more than one default poller is defined in the application context. Any endpoints connected to a `PollableChannel` (`PollingConsumer`) or any `SourcePollingChannelAdapter` that does not have an explicitly configured poller then uses the global default poller. It defaults to `false`. Optional.

③ Identifies the channel to which error messages are sent if a failure occurs in this poller's invocation. To completely suppress exceptions, you can provide a reference to the `nullChannel`. Optional.

④ The fixed delay trigger uses a `PeriodicTrigger` under the covers. If you do not use the `time-unit` attribute, the specified value is represented in milliseconds. If this attribute is set, none of the following attributes must be specified: `fixed-rate`, `trigger`, `cron`, and `ref`.

⑤ The fixed rate trigger uses a `PeriodicTrigger` under the covers. If you do not use the `time-unit` attribute, the specified value is represented in milliseconds. If this attribute is set, none of the following attributes must be specified: `fixed-delay`, `trigger`, `cron`, and `ref`.

⑥ The ID referring to the poller's underlying bean-definition, which is of type `org.springframework.integration.scheduling.PollerMetadata`. The `id` attribute is required for a top-level poller element, unless it is the default poller (`default="true"`).

⑦ See Configuring An Inbound Channel Adapter for more information. If not specified, the default value depends on the context. If you use a `PollingConsumer`, this attribute defaults to `-1`.

However, if you use a `SourcePollingChannelAdapter`, the `max-messages-per-poll` attribute defaults to `1`. Optional.

⑧ Value is set on the underlying class `PollerMetadata`. If not specified, it defaults to 1000 (milliseconds). Optional.

⑨ Bean reference to another top-level poller. The `ref` attribute must not be present on the top-level `poller` element. However, if this attribute is set, none of the following attributes must be specified: `fixed-rate`, `trigger`, `cron`, and `fixed-delay`.

⑩ Provides the ability to reference a custom task executor. See TaskExecutor Support for further information. Optional.

⑪ This attribute specifies the `java.util.concurrent.TimeUnit` enum value on the underlying `org.springframework.scheduling.support.PeriodicTrigger`. Therefore, this attribute can be used only in combination with the `fixed-delay` or `fixed-rate` attributes. If combined with either `cron` or a `trigger` reference attribute, it causes a failure. The minimal supported granularity for a `PeriodicTrigger` is milliseconds. Therefore, the only available options are milliseconds and seconds. If this value is not provided, any `fixed-delay` or `fixed-rate` value is interpreted as milliseconds. Basically, this enum provides a convenience for seconds-based interval trigger values. For hourly, daily, and monthly settings, we recommend using a `cron` trigger instead.

⑫ Reference to any Spring-configured bean that implements the `org.springframework.scheduling.Trigger` interface. However, if this attribute is set, none of the following attributes must be specified: `fixed-delay`, `fixed-rate`, `cron`, and `ref`. Optional.

⑬ Allows specifying extra AOP advices to handle additional cross-cutting concerns. See Transaction Support for further information. Optional.

⑭ Pollers can be made transactional. See AOP Advice chains for further information. Optional.

Examples

A simple interval-based poller with a 1-second interval can be configured as follows:

```
<int:transformer input-channel="pollable"
    ref="transformer"
    output-channel="output">
    <int:poller fixed-rate="1000"/>
</int:transformer>
```

As an alternative to using the `fixed-rate` attribute, you can also use the `fixed-delay` attribute.

For a poller based on a Cron expression, use the `cron` attribute instead, as the following example shows:

```
<int:transformer input-channel="pollable"
    ref="transformer"
    output-channel="output">
    <int:poller cron="*/10 * * * * MON-FRI"/>
</int:transformer>
```

If the input channel is a `PollableChannel`, the poller configuration is required. Specifically, as mentioned earlier, the `trigger` is a required property of the `PollingConsumer` class. Therefore, if you omit the `poller` sub-element for a polling consumer endpoint's configuration, an exception may be thrown. The exception may also be thrown if you attempt to configure a poller on the element that is connected to a non-pollable channel.

It is also possible to create top-level pollers, in which case only a `ref` attribute is required, as the following example shows:

```
<int:poller id="weekdayPoller" cron="*/10 * * * * MON-FRI"/>

<int:transformer input-channel="pollable"
    ref="transformer"
    output-channel="output">
    <int:poller ref="weekdayPoller"/>
</int:transformer>
```

The `ref` attribute is allowed only on the inner poller definitions. Defining this attribute on a top-level poller results in a configuration exception being thrown during initialization of the application context.

Global Default Pollers

To simplify the configuration even further, you can define a global default poller. A single top-level poller within an `ApplicationContext` may have the `default` attribute set to `true`. In that case, any endpoint with a `PollableChannel` for its input channel, that is defined within the same `ApplicationContext`, and has no explicitly configured `poller` sub-element uses that default. The following example shows such a poller and a transformer that uses it:

```
<int:poller id="defaultPoller" default="true" max-messages-per-poll="5" fixed-rate=
"3000"/>

<!-- No <poller/> sub-element is necessary, because there is a default -->
<int:transformer input-channel="pollable"
                 ref="transformer"
                 output-channel="output"/>
```

Transaction Support

Spring Integration also provides transaction support for the pollers so that each receive-and-forward operation can be performed as an atomic unit of work. To configure transactions for a poller, add the `<transactional/>` sub-element. The following example shows the available attributes:

```
<int:poller fixed-delay="1000">
    <int:transactional transaction-manager="txManager"
                       propagation="REQUIRED"
                       isolation="REPEATABLE_READ"
                       timeout="10000"
                       read-only="false"/>
</int:poller>
```

For more information, see Poller Transaction Support.

AOP Advice chains

Since Spring transaction support depends on the proxy mechanism with `TransactionInterceptor` (AOP Advice) handling transactional behavior of the message flow initiated by the poller, you must sometimes provide extra advices to handle other cross cutting behavior associated with the poller. For that, the `poller` defines an `advice-chain` element that lets you add more advices in a class that implements the `MethodInterceptor` interface. The following example shows how to define an `advice-chain` for a `poller`:

```
<int:service-activator id="advicedSa" input-channel="goodInputWithAdvice" ref=
"testBean"
        method="good" output-channel="output">
    <int:poller max-messages-per-poll="1" fixed-rate="10000">
         <int:advice-chain>
            <ref bean="adviceA" />
            <beans:bean class="org.something.SampleAdvice" />
            <ref bean="txAdvice" />
        </int:advice-chain>
    </int:poller>
</int:service-activator>
```

For more information on how to implement the `MethodInterceptor` interface, see the AOP sections of the Spring Framework Reference Guide. An advice chain can also be applied on a poller that does not have any transaction configuration, letting you enhance the behavior of the message flow initiated by the poller.

When using an advice chain, the `<transactional/>` child element cannot be specified. Instead, declare a `<tx:advice/>` bean and add it to the `<advice-chain/>`. See Poller Transaction Support for complete configuration details.

TaskExecutor Support

The polling threads may be executed by any instance of Spring's `TaskExecutor` abstraction. This enables concurrency for an endpoint or group of endpoints. As of Spring 3.0, the core Spring Framework has a `task` namespace, and its `<executor/>` element supports the creation of a simple thread pool executor. That element accepts attributes for common concurrency settings, such as pool-size and queue-capacity. Configuring a thread-pooling executor can make a substantial difference in how the endpoint performs under load. These settings are available for each endpoint, since the performance of an endpoint is one of the major factors to consider (the other major factor being the expected volume on the channel to which the endpoint subscribes). To enable concurrency for a polling endpoint that is configured with the XML namespace support, provide the `task-executor` reference on its `<poller/>` element and then provide one or more of the properties shown in the following example:

```
<int:poller task-executor="pool" fixed-rate="1000"/>

<task:executor id="pool"
               pool-size="5-25"
               queue-capacity="20"
               keep-alive="120"/>
```

If you do not provide a task-executor, the consumer's handler is invoked in the caller's thread. Note that the caller is usually the default `TaskScheduler` (see Configuring the Task Scheduler). You should also keep in mind that the `task-executor` attribute can provide a reference to any implementation of Spring's `TaskExecutor` interface by specifying the bean name. The `executor` element shown earlier is provided for convenience.

As mentioned earlier in the background section for polling consumers, you can also configure a polling consumer in such a way as to emulate event-driven behavior. With a long `receive-timeout` and a short `interval-trigger`, you can ensure a very timely reaction to arriving messages even on a polled message source. Note that this applies only to sources that have a blocking wait call with a timeout. For example, the file poller does not block. Each receive() call returns immediately and either contains new files or not. Therefore, even if a poller contains a long `receive-timeout`, that value would never be used in such a scenario. On the other hand, when using Spring Integration's own queue-based channels, the timeout value does have a chance to participate. The following example shows how a polling consumer can receive messages nearly instantaneously:

```
<int:service-activator input-channel="someQueueChannel"
    output-channel="output">
    <int:poller receive-timeout="30000" fixed-rate="10"/>

</int:service-activator>
```

Using this approach does not carry much overhead, since, internally, it is nothing more then a

timed-wait thread, which does not require nearly as much CPU resource usage as (for example) a thrashing, infinite while loop.

10.1.5. Changing Polling Rate at Runtime

When configuring a poller with a `fixed-delay` or a `fixed-rate` attribute, the default implementation uses a `PeriodicTrigger` instance. The `PeriodicTrigger` is part of the core Spring Framework. It accepts the interval only as a constructor argument. Therefore, it cannot be changed at runtime.

However, you can define your own implementation of the `org.springframework.scheduling.Trigger` interface. You could even use the `PeriodicTrigger` as a starting point. Then you can add a setter for the interval (period), or you can even embed your own throttling logic within the trigger itself. The `period` property is used with each call to `nextExecutionTime` to schedule the next poll. To use this custom trigger within pollers, declare the bean definition of the custom trigger in your application context and inject the dependency into your poller configuration by using the `trigger` attribute, which references the custom trigger bean instance. You can now obtain a reference to the trigger bean and change the polling interval between polls.

For an example, see the Spring Integration Samples project. It contains a sample called `dynamic-poller`, which uses a custom trigger and demonstrates the ability to change the polling interval at runtime.

The sample provides a custom trigger that implements the `org.springframework.scheduling.Trigger` interface. The sample's trigger is based on Spring's `PeriodicTrigger` implementation. However, the fields of the custom trigger are not final, and the properties have explicit getters and setters, letting you dynamically change the polling period at runtime.

It is important to note, though, that because the Trigger method is `nextExecutionTime()`, any changes to a dynamic trigger do not take effect until the next poll, based on the existing configuration. It is not possible to force a trigger to fire before its currently configured next execution time.

10.1.6. Payload Type Conversion

Throughout this reference manual, you can also see specific configuration and implementation examples of various endpoints that accept a message or any arbitrary `Object` as an input parameter. In the case of an `Object`, such a parameter is mapped to a message payload or part of the payload or header (when using the Spring Expression Language). However, the type of input parameter of the endpoint method sometimes does not match the type of the payload or its part. In this scenario, we need to perform type conversion. Spring Integration provides a convenient way for registering type converters (by using the Spring `ConversionService`) within its own instance of a conversion service bean named `integrationConversionService`. That bean is automatically created as soon as the first converter is defined by using the Spring Integration infrastructure. To register a converter, you can implement `org.springframework.core.convert.converter.Converter`, `org.springframework.core.convert.converter.GenericConverter`, or `org.springframework.core.convert.converter.ConverterFactory`.

The `Converter` implementation is the simplest and converts from a single type to another. For more sophistication, such as converting to a class hierarchy, you can implement a `GenericConverter` and

possibly a `ConditionalConverter`. These give you complete access to the `from` and `to` type descriptors, enabling complex conversions. For example, if you have an abstract class called `Something` that is the target of your conversion (parameter type, channel data type, and so on), you have two concrete implementations called `Thing1` and `Thing`, and you wish to convert to one or the other based on the input type, the `GenericConverter` would be a good fit. For more information, see the Javadoc for these interfaces:

- org.springframework.core.convert.converter.Converter
- org.springframework.core.convert.converter.GenericConverter
- org.springframework.core.convert.converter.ConverterFactory

When you have implemented your converter, you can register it with convenient namespace support, as the following example shows:

```
<int:converter ref="sampleConverter"/>

<bean id="sampleConverter" class="foo.bar.TestConverter"/>
```

Alternately, you can use an inner bean, as the following example shows:

```
<int:converter>
    <bean class="o.s.i.config.xml.ConverterParserTests$TestConverter3"/>
</int:converter>
```

Starting with Spring Integration 4.0, you can use annotations to create the preceding configuration, as the following example shows:

```
@Component
@IntegrationConverter
public class TestConverter implements Converter<Boolean, Number> {

    public Number convert(Boolean source) {
        return source ? 1 : 0;
    }

}
```

Alternately, you can use the `@Configuration` annotation, as the following example shows:

```
@Configuration
@EnableIntegration
public class ContextConfiguration {

    @Bean
    @IntegrationConverter
    public SerializingConverter serializingConverter() {
        return new SerializingConverter();
    }

}
```

When configuring an application context, the Spring Framework lets you add a `conversionService` bean (see Configuring a ConversionService chapter). This service is used, when needed, to perform appropriate conversions during bean creation and configuration.

In contrast, the `integrationConversionService` is used for runtime conversions. These uses are quite different. Converters that are intended for use when wiring bean constructor arguments and properties may produce unintended results if used at runtime for Spring Integration expression evaluation against messages within data type channels, payload type transformers, and so on.

However, if you do want to use the Spring `conversionService` as the Spring Integration `integrationConversionService`, you can configure an alias in the application context, as the following example shows:

```
<alias name="conversionService" alias="
integrationConversionService"/>
```

In this case, the converters provided by the `conversionService` are available for Spring Integration runtime conversion.

10.1.7. Content Type Conversion

Starting with version 5.0, by default, the method invocation mechanism is based on the `org.springframework.messaging.handler.invocation.InvocableHandlerMethod` infrastructure. Its `HandlerMethodArgumentResolver` implementations (such as `PayloadArgumentResolver` and `MessageMethodArgumentResolver`) can use the `MessageConverter` abstraction to convert an incoming `payload` to the target method argument type. The conversion can be based on the `contentType` message header. For this purpose, Spring Integration provides the `ConfigurableCompositeMessageConverter`, which delegates to a list of registered converters to be invoked until one of them returns a non-null result. By default, this converter provides (in strict order):

1. `MappingJackson2MessageConverter` if the Jackson processor is present on the classpath
2. `ByteArrayMessageConverter`
3. `ObjectStringMessageConverter`
4. `GenericMessageConverter`

See the Javadoc (linked in the preceding list) for more information about their purpose and appropriate `contentType` values for conversion. The `ConfigurableCompositeMessageConverter` is used because it can be be supplied with any other `MessageConverter` implementations, including or excluding the previously mentioned default converters. It can also be registered as an appropriate bean in the application context, overriding the default converter, as the following example shows:

```
@Bean(name = IntegrationContextUtils.
ARGUMENT_RESOLVER_MESSAGE_CONVERTER_BEAN_NAME)
public ConfigurableCompositeMessageConverter compositeMessageConverter() {
    List<MessageConverter> converters =
        Arrays.asList(new MarshallingMessageConverter(jaxb2Marshaller()),
                  new JavaSerializationMessageConverter());
    return new ConfigurableCompositeMessageConverter(converters);
}
```

Those two new converters are registered in the composite before the defaults. You can also not use a `ConfigurableCompositeMessageConverter` but provide your own `MessageConverter` by registering a bean with the name, `integrationArgumentResolverMessageConverter` (by setting the `IntegrationContextUtils.ARGUMENT_RESOLVER_MESSAGE_CONVERTER_BEAN_NAME` property).

The `MessageConverter`-based (including `contentType` header) conversion is not available when using SpEL method invocation. In this case, only the regular class-to-class conversion mentioned above in the Payload Type Conversion is available.

10.1.8. Asynchronous Polling

If you want the polling to be asynchronous, a poller can optionally specify a `task-executor` attribute that points to an existing instance of any `TaskExecutor` bean (Spring 3.0 provides a convenient namespace configuration through the `task` namespace). However, there are certain things you must understand when configuring a poller with a `TaskExecutor`.

The problem is that there are two configurations in place, the poller and the `TaskExecutor`. They must be in tune with each other. Otherwise, you might end up creating an artificial memory leak.

Consider the following configuration:

```
<int:channel id="publishChannel">
    <int:queue />
</int:channel>

<int:service-activator input-channel="publishChannel" ref="myService">
    <int:poller receive-timeout="5000" task-executor="taskExecutor" fixed-rate="
50" />
</int:service-activator>

<task:executor id="taskExecutor" pool-size="20" />
```

The preceding configuration demonstrates an out-of-tune configuration.

By default, the task executor has an unbounded task queue. The poller keeps scheduling new tasks even though all the threads are blocked, waiting for either a new message to arrive or the timeout to expire. Given that there are 20 threads executing tasks with a five-second timeout, they are executed at a rate of 4 per second. However, new tasks are being scheduled at a rate of 20 per second, so the internal queue in the task executor grows at a rate of 16 per second (while the process is idle), so we have a memory leak.

One of the ways to handle this is to set the `queue-capacity` attribute of the task executor. Even 0 is a reasonable value. You can also manage it by specifying what to do with messages that can not be queued by setting the `rejection-policy` attribute of the Task Executor (for example, to `DISCARD`). In other words, there are certain details you must understand when configuring `TaskExecutor`. See "Task Execution and Scheduling" in the Spring reference manual for more detail on the subject.

10.1.9. Endpoint Inner Beans

Many endpoints are composite beans. This includes all consumers and all polled inbound channel adapters. Consumers (polled or event-driven) delegate to a `MessageHandler`. Polled adapters obtain messages by delegating to a `MessageSource`. Often, it is useful to obtain a reference to the delegate bean, perhaps to change configuration at runtime or for testing. These beans can be obtained from the `ApplicationContext` with well known names. `MessageHandler` instances are registered with the application context with bean IDs similar to `someConsumer.handler` (where 'consumer' is the value of the endpoint's `id` attribute). `MessageSource` instances are registered with bean IDs similar to `somePolledAdapter.source`, where 'somePolledAdapter' is the ID of the adapter.

The preceding only applies to the framework component itself. You can instead use an inner bean definition, as the following example shows:

```
<int:service-activator id="exampleServiceActivator" input-channel="inChannel"
            output-channel = "outChannel" method="foo">
    <beans:bean class="org.foo.ExampleServiceActivator"/>
</int:service-activator>
```

The bean is treated like any inner bean declared and is not registered with the application context. If you wish to access this bean in some other manner, declare it at the top level with an `id` and use the `ref` attribute instead. See the Spring Documentation for more information.

10.2. Endpoint Roles

Starting with version 4.2, endpoints can be assigned to roles. Roles let endpoints be started and stopped as a group. This is particularly useful when using leadership election, where a set of endpoints can be started or stopped when leadership is granted or revoked, respectively. For this purpose the framework registers a `SmartLifecycleRoleController` bean in the application context with the name `IntegrationContextUtils.INTEGRATION_LIFECYCLE_ROLE_CONTROLLER`. Whenever it is necessary to control lifecycles, this bean can be injected or `@Autowired`:

```
<bean class="com.some.project.SomeLifecycleControl">
    <property name="roleController" ref="integrationLifecycleRoleController"/>
</bean>
```

You can assign endpoints to roles using XML, Java configuration, or programmatically. The following example shows how to configure endpoint roles with XML:

```
<int:inbound-channel-adapter id="ica" channel="someChannel" expression="'foo'"
role="cluster"
        auto-startup="false">
    <int:poller fixed-rate="60000" />
</int:inbound-channel-adapter>
```

The following example shows how to configure endpoint roles for a bean created in Java:

```
@Bean
@ServiceActivator(inputChannel = "sendAsyncChannel", autoStartup="false")
@Role("cluster")
public MessageHandler sendAsyncHandler() {
    return // some MessageHandler
}
```

The following example shows how to configure endpoint roles on a method in Java:

```
@Payload("#args[0].toLowerCase()")
@Role("cluster")
public String handle(String payload) {
    return payload.toUpperCase();
}
```

The following example shows how to configure endpoint roles by using the `SmartLifecycleRoleController` in Java:

```
@Autowired
private SmartLifecycleRoleController roleController;
...
    this.roleController.addSmartLifeCycleToRole("cluster", someEndpoint);
...
```

The following example shows how to configure endpoint roles by using an `IntegrationFlow` in Java:

```
IntegrationFlow flow -> flow
        .handle(..., e -> e.role("cluster"));
```

Each of these adds the endpoint to the `cluster` role.

Invoking `roleController.startLifecyclesInRole("cluster")` and the corresponding `stop…` method starts and stops the endpoints.

Any object that implements `SmartLifecycle` can be programmatically added — not just endpoints.

The `SmartLifecycleRoleController` implements `ApplicationListener<AbstractLeaderEvent>` and it automatically starts and stops its configured `SmartLifecycle` objects when leadership is granted or revoked (when some bean publishes `OnGrantedEvent` or `OnRevokedEvent`, respectively).

When using leadership election to start and stop components, it is important to set the `auto-startup` XML attribute (`autoStartup` bean property) to `false` so that the application context does not start the components during context initialization.

Starting with version 4.3.8, the `SmartLifecycleRoleController` provides several status methods:

```
public Collection<String> getRoles() ①

public boolean allEndpointsRunning(String role) ②

public boolean noEndpointsRunning(String role) ③

public Map<String, Boolean> getEndpointsRunningStatus(String role) ④
```

① Returns a list of the roles being managed.

② Returns `true` if all endpoints in the role are running.

③ Returns `true` if none of the endpoints in the role are running.

④ Returns a map of `component name : running status`. The component name is usually the bean name.

10.3. Leadership Event Handling

Groups of endpoints can be started and stopped based on leadership being granted or revoked, respectively. This is useful in clustered scenarios where shared resources must be consumed by only a single instance. An example of this is a file inbound channel adapter that is polling a shared directory. (See Reading Files).

To participate in a leader election and be notified when elected leader, when leadership is revoked, or on failure to acquire the resources to become leader, an application creates a component in the application context called a “leader initiator”. Normally, a leader initiator is a `SmartLifecycle`, so it starts (optionally) when the context starts and then publishes notifications when leadership changes. You can also receive failure notifications by setting the `publishFailedEvents` to `true` (starting with version 5.0), for cases when you want take a specific action if a failure occurs. By convention, you should provide a `Candidate` that receives the callbacks. You can also revoke the leadership through a `Context` object provided by the framework. Your code can also listen for `o.s.i.leader.event.AbstractLeaderEvent` instances (the super class of `OnGrantedEvent` and `OnRevokedEvent`) and respond accordingly (for instance, by using a `SmartLifecycleRoleController`). The events contain a reference to the `Context` object. The following listing shows the definition of the `Context` interface:

```
public interface Context {

    boolean isLeader();

    void yield();

    String getRole();

}
```

Starting with version 5.0.6, the context provides a reference to the candidate's role.

Spring Integration provides a basic implementation of a leader initiator that is based on the `LockRegistry` abstraction. To use it, you need to create an instance as a bean, as the following example shows:

```
@Bean
public LockRegistryLeaderInitiator leaderInitiator(LockRegistry locks) {
    return new LockRegistryLeaderInitiator(locks);
}
```

If the lock registry is implemented correctly, there is only ever at most one leader. If the lock registry also provides locks that throw exceptions (ideally, `InterruptedException`) when they expire or are broken, the duration of the leaderless periods can be as short as is allowed by the inherent latency in the lock implementation. By default, the `busyWaitMillis` property adds some additional latency to prevent CPU starvation in the (more usual) case that the locks are imperfect and you only know they expired when you try to obtain one again.

See Zookeeper Leadership Event Handling for more information about leadership election and events that use Zookeeper.

10.4. Messaging Gateways

A gateway hides the messaging API provided by Spring Integration. It lets your application's business logic be unaware of the Spring Integration API. By using a generic Gateway, your code interacts with only a simple interface.

10.4.1. Enter the `GatewayProxyFactoryBean`

As mentioned earlier, it would be great to have no dependency on the Spring Integration API — including the gateway class. For that reason, Spring Integration provides the `GatewayProxyFactoryBean`, which generates a proxy for any interface and internally invokes the gateway methods shown below. By using dependency injection, you can then expose the interface to your business methods.

The following example shows an interface that can be used to interact with Spring Integration:

```
package org.cafeteria;

public interface Cafe {

    void placeOrder(Order order);

}
```

10.4.2. Gateway XML Namespace Support

Namespace support is also provided. It lets you configure an interface as a service, as the following example shows:

```
<int:gateway id="cafeService"
         service-interface="org.cafeteria.Cafe"
         default-request-channel="requestChannel"
         default-reply-timeout="10000"
         default-reply-channel="replyChannel"/>
```

With this configuration defined, the `cafeService` can now be injected into other beans, and the code that invokes the methods on that proxied instance of the `Cafe` interface has no awareness of the Spring Integration API. The general approach is similar to that of Spring Remoting (RMI, HttpInvoker, and so on). See the "Samples" Appendix for an example that uses the `gateway` element (in the Cafe demo).

The defaults in the preceding configuration are applied to all methods on the gateway interface. If a reply timeout is not specified, the calling thread waits indefinitely for a reply. See Gateway Behavior When No response Arrives.

The defaults can be overridden for individual methods. See Gateway Configuration with Annotations and XML.

10.4.3. Setting the Default Reply Channel

Typically, you need not specify the `default-reply-channel`, since a Gateway auto-creates a temporary, anonymous reply channel, where it listens for the reply. However, some cases may prompt you to define a `default-reply-channel` (or `reply-channel` with adapter gateways, such as HTTP, JMS, and others).

For some background, we briefly discuss some of the inner workings of the gateway. A gateway creates a temporary point-to-point reply channel. It is anonymous and is added to the message headers with the name, `replyChannel`. When providing an explicit `default-reply-channel` (`reply-`

`channel` with remote adapter gateways), you can point to a publish-subscribe channel, which is so named because you can add more than one subscriber to it. Internally, Spring Integration creates a bridge between the temporary `replyChannel` and the explicitly defined `default-reply-channel`.

Suppose you want your reply to go not only to the gateway but also to some other consumer. In this case, you want two things:

- A named channel to which you can subscribe
- That channel to be a publish-subscribe-channel

The default strategy used by the gateway does not satisfy those needs, because the reply channel added to the header is anonymous and point-to-point. This means that no other subscriber can get a handle to it and, even if it could, the channel has point-to-point behavior such that only one subscriber would get the message. By defining a `default-reply-channel` you can point to a channel of your choosing. In this case, that is a `publish-subscribe-channel`. The gateway creates a bridge from it to the temporary, anonymous reply channel that is stored in the header.

You might also want to explicitly provide a reply channel for monitoring or auditing through an interceptor (for example, wiretap). To configure a channel interceptor, you need a named channel.

10.4.4. Gateway Configuration with Annotations and XML

Consider the following example, which expands on the previous `Cafe` interface example by adding a `@Gateway` annotation:

```
public interface Cafe {

    @Gateway(requestChannel="orders")
    void placeOrder(Order order);

}
```

The `@Header` annotation lets you add values that are interpreted as message headers, as the following example shows:

```
public interface FileWriter {

    @Gateway(requestChannel="filesOut")
    void write(byte[] content, @Header(FileHeaders.FILENAME) String filename);

}
```

If you prefer the XML approach to configuring gateway methods, you can add `method` elements to the gateway configuration, as the following example shows:

```
<int:gateway id="myGateway" service-interface="org.foo.bar.TestGateway"
      default-request-channel="inputC">
  <int:default-header name="calledMethod" expression="#gatewayMethod.name"/>
  <int:method name="echo" request-channel="inputA" reply-timeout="2" request-
timeout="200"/>
  <int:method name="echoUpperCase" request-channel="inputB"/>
  <int:method name="echoViaDefault"/>
</int:gateway>
```

You can also use XML to provide individual headers for each method invocation. This could be useful if the headers you want to set are static in nature and you do not want to embed them in the gateway's method signature by using `@Header` annotations. For example, in the loan broker example, we want to influence how aggregation of the loan quotes is done, based on what type of request was initiated (single quote or all quotes). Determining the type of the request by evaluating which gateway method was invoked, although possible, would violate the separation of concerns paradigm (the method is a Java artifact). However, expressing your intention (meta information) in message headers is natural in a messaging architecture. The following example shows how to add a different message header for each of two methods:

```
<int:gateway id="loanBrokerGateway"
         service-interface=
"org.springframework.integration.loanbroker.LoanBrokerGateway">
  <int:method name="getLoanQuote" request-channel="loanBrokerPreProcessingChannel
">
    <int:header name="RESPONSE_TYPE" value="BEST"/>
  </int:method>
  <int:method name="getAllLoanQuotes" request-channel=
"loanBrokerPreProcessingChannel">
    <int:header name="RESPONSE_TYPE" value="ALL"/>
  </int:method>
</int:gateway>
```

In the preceding example a different value is set for the 'RESPONSE_TYPE' header, based on the gateway's method.

Expressions and "Global" Headers

The `<header/>` element supports `expression` as an alternative to `value`. The SpEL expression is evaluated to determine the value of the header. Starting with version 5.2, the `#root` object of the evaluation context is a `MethodArgsHolder` with `getMethod()` and `getArgs()` accessors.

These two expression evaluation context variables are deprecated since version 5.2:

- #args: An `Object[]` containing the method arguments

- #gatewayMethod: The object (derived from `java.reflect.Method`) that represents the method in the `service-interface` that was invoked. A header containing this variable can be used later in the flow (for example, for routing). For example, if you wish to route on the simple method name, you might add a header with the following expression: `#gatewayMethod.name`.

The `java.reflect.Method` is not serializable. A header with an expression of `method` is lost if you later serialize the message. Consequently, you may wish to use `method.name` or `method.toString()` in those cases. The `toString()` method provides a `String` representation of the method, including parameter and return types.

Since version 3.0, `<default-header/>` elements can be defined to add headers to all the messages produced by the gateway, regardless of the method invoked. Specific headers defined for a method take precedence over default headers. Specific headers defined for a method here override any `@Header` annotations in the service interface. However, default headers do NOT override any `@Header` annotations in the service interface.

The gateway now also supports a `default-payload-expression`, which is applied for all methods (unless overridden).

10.4.5. Mapping Method Arguments to a Message

Using the configuration techniques in the previous section allows control of how method arguments are mapped to message elements (payload and headers). When no explicit configuration is used, certain conventions are used to perform the mapping. In some cases, these conventions cannot determine which argument is the payload and which should be mapped to headers. Consider the following example:

```
public String send1(Object thing1, Map thing2);

public String send2(Map thing1, Map thing2);
```

In the first case, the convention is to map the first argument to the payload (as long as it is not a `Map`) and the contents of the second argument become headers.

In the second case (or the first when the argument for parameter `thing1` is a `Map`), the framework cannot determine which argument should be the payload. Consequently, mapping fails. This can generally be resolved using a `payload-expression`, a `@Payload` annotation, or a `@Headers` annotation.

Alternatively (and whenever the conventions break down), you can take the entire responsibility for mapping the method calls to messages. To do so, implement an `MethodArgsMessageMapper` and provide it to the `<gateway/>` by using the `mapper` attribute. The mapper maps a `MethodArgsHolder`, which is a simple class that wraps the `java.reflect.Method` instance and an `Object[]` containing the arguments. When providing a custom mapper, the `default-payload-expression` attribute and `<default-header/>` elements are not allowed on the gateway. Similarly, the `payload-expression` attribute and `<header/>` elements are not allowed on any `<method/>` elements.

Mapping Method Arguments

The following examples show how method arguments can be mapped to the message and shows some examples of invalid configuration:

```
public interface MyGateway {

    void payloadAndHeaderMapWithoutAnnotations(String s, Map<String, Object> map);

    void payloadAndHeaderMapWithAnnotations(@Payload String s, @Headers Map<
String, Object> map);

    void headerValuesAndPayloadWithAnnotations(@Header("k1") String x, @Payload
String s, @Header("k2") String y);

    void mapOnly(Map<String, Object> map); // the payload is the map and no custom
headers are added

    void twoMapsAndOneAnnotatedWithPayload(@Payload Map<String, Object> payload,
Map<String, Object> headers);

    @Payload("#args[0] + #args[1] + '!'")
    void payloadAnnotationAtMethodLevel(String a, String b);

    @Payload("@someBean.exclaim(#args[0])")
    void payloadAnnotationAtMethodLevelUsingBeanResolver(String s);

    void payloadAnnotationWithExpression(@Payload("toUpperCase()") String s);

    void payloadAnnotationWithExpressionUsingBeanResolver(@Payload(
"@someBean.sum(#this)") String s); //  ①

    // invalid
    void twoMapsWithoutAnnotations(Map<String, Object> m1, Map<String, Object> m2
);

    // invalid
    void twoPayloads(@Payload String s1, @Payload String s2);

    // invalid
    void payloadAndHeaderAnnotationsOnSameParameter(@Payload @Header("x") String
s);

    // invalid
    void payloadAndHeadersAnnotationsOnSameParameter(@Payload @Headers Map<String,
Object> map);

}
```

① Note that, in this example, the SpEL variable, `#this`, refers to the argument — in this case, the value of `s`.

The XML equivalent looks a little different, since there is no `#this` context for the method argument.

However, expressions can refer to method arguments by using the `#args` variable, as the following example shows:

```
<int:gateway id="myGateway" service-interface="org.something.MyGateway">
  <int:method name="send1" payload-expression="#args[0] + 'thing2'"/>
  <int:method name="send2" payload-expression="@someBean.sum(#args[0])"/>
  <int:method name="send3" payload-expression="#method"/>
  <int:method name="send4">
    <int:header name="thing1" expression="#args[2].toUpperCase()"/>
  </int:method>
</int:gateway>
```

10.4.6. `@MessagingGateway` Annotation

Starting with version 4.0, gateway service interfaces can be marked with a `@MessagingGateway` annotation instead of requiring the definition of a `<gateway />` xml element for configuration. The following pair of examples compares the two approaches for configuring the same gateway:

```
<int:gateway id="myGateway" service-interface="org.something.TestGateway"
      default-request-channel="inputC">
  <int:default-header name="calledMethod" expression="#gatewayMethod.name"/>
  <int:method name="echo" request-channel="inputA" reply-timeout="2" request-
timeout="200"/>
  <int:method name="echoUpperCase" request-channel="inputB">
    <int:header name="thing1" value="thing2"/>
  </int:method>
  <int:method name="echoViaDefault"/>
</int:gateway>
```

```
@MessagingGateway(name = "myGateway", defaultRequestChannel = "inputC",
          defaultHeaders = @GatewayHeader(name = "calledMethod",
                                    expression="#gatewayMethod.name"))
public interface TestGateway {

   @Gateway(requestChannel = "inputA", replyTimeout = 2, requestTimeout = 200)
   String echo(String payload);

   @Gateway(requestChannel = "inputB", headers = @GatewayHeader(name = "thing1",
value="thing2"))
   String echoUpperCase(String payload);

   String echoViaDefault(String payload);

}
```

Similarly to the XML version, when Spring Integration discovers these annotations during a component scan, it creates the `proxy` implementation with its messaging infrastructure. To perform this scan and register the `BeanDefinition` in the application context, add the `@IntegrationComponentScan` annotation to a `@Configuration` class. The standard `@ComponentScan` infrastructure does not deal with interfaces. Consequently, we introduced the custom `@IntegrationComponentScan` logic to fine the `@MessagingGateway` annotation on the interfaces and register `GatewayProxyFactoryBean` instances for them. See also Annotation Support.

Along with the `@MessagingGateway` annotation you can mark a service interface with the `@Profile` annotation to avoid the bean creation, if such a profile is not active.

If you have no XML configuration, the `@EnableIntegration` annotation is required on at least one `@Configuration` class. See Configuration and `@EnableIntegration` for more information.

10.4.7. Invoking No-Argument Methods

When invoking methods on a Gateway interface that do not have any arguments, the default behavior is to receive a `Message` from a `PollableChannel`.

Sometimes, however, you may want to trigger no-argument methods so that you can interact with other components downstream that do not require user-provided parameters, such as triggering no-argument SQL calls or stored procedures.

To achieve send-and-receive semantics, you must provide a payload. To generate a payload, method parameters on the interface are not necessary. You can either use the `@Payload` annotation or the `payload-expression` attribute in XML on the `method` element. The following list includes a few examples of what the payloads could be:

- a literal string
- #gatewayMethod.name
- new java.util.Date()
- @someBean.someMethod()'s return value

The following example shows how to use the `@Payload` annotation:

```
public interface Cafe {

    @Payload("new java.util.Date()")
    List<Order> retrieveOpenOrders();

}
```

If a method has no argument and no return value but does contain a payload expression, it is treated as a send-only operation.

10.4.8. Invoking `default` Methods

An interface for gateway proxy may have `default` methods as well and starting with version 5.3, the framework injects a `DefaultMethodInvokingMethodInterceptor` into a proxy for calling `default` methods using a `java.lang.invoke.MethodHandle` approach instead of proxying. The interfaces from JDK, such as `java.util.function.Function`, still can be used for gateway proxy, but their `default` methods cannot be called because of internal Java security reasons for a `MethodHandles.Lookup` instantiation against JDK classes. These methods also can be proxied (losing their implementation logic and, at the same time, restoring previous gateway proxy behavior) using an explicit `@Gateway` annotation on the method, or `proxyDefaultMethods` on the `@MessagingGateway` annotation or `<gateway>` XML component.

10.4.9. Error Handling

The gateway invocation can result in errors. By default, any error that occurs downstream is re-thrown “as is” upon the gateway’s method invocation. For example, consider the following simple

flow:

```
gateway -> service-activator
```

If the service invoked by the service activator throws a `MyException` (for example), the framework wraps it in a `MessagingException` and attaches the message passed to the service activator in the `failedMessage` property. Consequently, any logging performed by the framework has full the context of the failure. By default, when the exception is caught by the gateway, the `MyException` is unwrapped and thrown to the caller. You can configure a `throws` clause on the gateway method declaration to match the particular exception type in the cause chain. For example, if you want to catch a whole `MessagingException` with all the messaging information of the reason of downstream error, you should have a gateway method similar to the following:

```
public interface MyGateway {

    void performProcess() throws MessagingException;

}
```

Since we encourage POJO programming, you may not want to expose the caller to messaging infrastructure.

If your gateway method does not have a `throws` clause, the gateway traverses the cause tree, looking for a `RuntimeException` that is not a `MessagingException`. If none is found, the framework throws the `MessagingException`. If the `MyException` in the preceding discussion has a cause of `SomeOtherException` and your method `throws SomeOtherException`, the gateway further unwraps that and throws it to the caller.

When a gateway is declared with no `service-interface`, an internal framework interface `RequestReplyExchanger` is used.

Consider the following example:

```
public interface RequestReplyExchanger {

    Message<?> exchange(Message<?> request) throws MessagingException;

}
```

Before version 5.0, this `exchange` method did not have a `throws` clause and, as a result, the exception was unwrapped. If you use this interface and want to restore the previous unwrap behavior, use a custom `service-interface` instead or access the `cause` of the `MessagingException` yourself.

However, you may want to log the error rather than propagating it or you may want to treat an

exception as a valid reply (by mapping it to a message that conforms to some "error message" contract that the caller understands). To accomplish this, the gateway provides support for a message channel dedicated to the errors by including support for the `error-channel` attribute. In the following example, a 'transformer' creates a reply `Message` from the `Exception`:

```
<int:gateway id="sampleGateway"
    default-request-channel="gatewayChannel"
    service-interface="foo.bar.SimpleGateway"
    error-channel="exceptionTransformationChannel"/>

<int:transformer input-channel="exceptionTransformationChannel"
        ref="exceptionTransformer" method="createErrorResponse"/>
```

The `exceptionTransformer` could be a simple POJO that knows how to create the expected error response objects. That becomes the payload that is sent back to the caller. You could do many more elaborate things in such an “error flow”, if necessary. It might involve routers (including Spring Integration’s `ErrorMessageExceptionTypeRouter`), filters, and so on. Most of the time, a simple 'transformer' should be sufficient, however.

Alternatively, you might want to only log the exception (or send it somewhere asynchronously). If you provide a one-way flow, nothing would be sent back to the caller. If you want to completely suppress exceptions, you can provide a reference to the global `nullChannel` (essentially a `/dev/null` approach). Finally, as mentioned above, if no `error-channel` is defined, then the exceptions propagate as usual.

When you use the `@MessagingGateway` annotation (see `@MessagingGateway Annotation`), you can use use the `errorChannel` attribute.

Starting with version 5.0, when you use a gateway method with a `void` return type (one-way flow), the `error-channel` reference (if provided) is populated in the standard `errorChannel` header of each sent message. This feature allows a downstream asynchronous flow, based on the standard `ExecutorChannel` configuration (or a `QueueChannel`), to override a default global `errorChannel` exceptions sending behavior. Previously you had to manually specify an `errorChannel` header with the `@GatewayHeader` annotation or the `<header>` element. The `error-channel` property was ignored for `void` methods with an asynchronous flow. Instead, error messages were sent to the default `errorChannel`.

Exposing the messaging system through simple POJI Gateways provides benefits, but “hiding” the reality of the underlying messaging system does come at a price, so there are certain things you should consider. We want our Java method to return as quickly as possible and not hang for an indefinite amount of time while the caller is waiting on it to return (whether void, a return value, or a thrown Exception). When regular methods are used as a proxies in front of the messaging system, we have to take into account the potentially asynchronous nature of the underlying messaging. This means that there might be a chance that a message that was initiated by a gateway could be dropped by a filter and never reach a component that is responsible for producing a reply. Some service activator method might result in an exception, thus providing no reply (as we do not generate null messages). In other words, multiple scenarios can cause a reply message to never come. That is perfectly natural in messaging systems. However, think about the implication on the gateway method. The gateway’s method input arguments were incorporated into a message and sent downstream. The reply message would be converted to a return value of the gateway’s method. So you might want to ensure that, for each gateway call, there is always a reply message. Otherwise, your gateway method might never return and hang indefinitely. One way to handle this situation is by using an asynchronous gateway (explained later in this section). Another way of handling it is to explicitly set the `reply-timeout` attribute. That way, the gateway does not hang any longer than the time specified by the `reply-timeout` and returns 'null' if that timeout does elapse. Finally, you might want to consider setting downstream flags, such as 'requires-reply', on a service-activator or 'throw-exceptions-on-rejection' on a filter. These options are discussed in more detail in the final section of this chapter.

If the downstream flow returns an `ErrorMessage`, its `payload` (a `Throwable`) is treated as a regular downstream error. If there is an `error-channel` configured, it is sent to the error flow. Otherwise the payload is thrown to the caller of the gateway. Similarly, if the error flow on the `error-channel` returns an `ErrorMessage`, its payload is thrown to the caller. The same applies to any message with a `Throwable` payload. This can be useful in asynchronous situations when when you need to propagate an `Exception` directly to the caller. To do so, you can either return an `Exception` (as the `reply` from some service) or throw it. Generally, even with an asynchronous flow, the framework takes care of propagating an exception thrown by the downstream flow back to the gateway. The TCP Client-Server Multiplex sample demonstrates both techniques to return the exception to the caller. It emulates a socket IO error to the waiting thread by using an `aggregator` with `group-timeout` (see Aggregator and Group Timeout) and a `MessagingTimeoutException` reply on the discard flow.

10.4.10. Gateway Timeouts

Gateways have two timeout properties: `requestTimeout` and `replyTimeout`. The request timeout applies only if the channel can block (for example, a bounded `QueueChannel` that is full). The `replyTimeout` value is how long the gateway waits for a reply or returns `null`. It defaults to infinity.

The timeouts can be set as defaults for all methods on the gateway (`defaultRequestTimeout` and `defaultReplyTimeout`) or on the `MessagingGateway` interface annotation. Individual methods can override these defaults (in `<method/>` child elements) or on the `@Gateway` annotation.

Starting with version 5.0, the timeouts can be defined as expressions, as the following example shows:

```
@Gateway(payloadExpression = "#args[0]", requestChannel = "someChannel",
        requestTimeoutExpression = "#args[1]", replyTimeoutExpression = "#args[2]
")
String lateReply(String payload, long requestTimeout, long replyTimeout);
```

The evaluation context has a `BeanResolver` (use `@someBean` to reference other beans), and the `#args` array variable is available.

When configuring with XML, the timeout attributes can be a long value or a SpEL expression, as the following example shows:

```
<method name="someMethod" request-channel="someRequestChannel"
                      payload-expression="#args[0]"
                      request-timeout="1000"
                      reply-timeout="#args[1]">
</method>
```

10.4.11. Asynchronous Gateway

As a pattern, the messaging gateway offers a nice way to hide messaging-specific code while still exposing the full capabilities of the messaging system. As described earlier, the `GatewayProxyFactoryBean` provides a convenient way to expose a proxy over a service-interface giving you POJO-based access to a messaging system (based on objects in your own domain, primitives/Strings, or other objects). However, when a gateway is exposed through simple POJO methods that return values, it implies that, for each request message (generated when the method is invoked), there must be a reply message (generated when the method has returned). Since messaging systems are naturally asynchronous, you may not always be able to guarantee the contract where "for each request, there will always be be a reply". Spring Integration 2.0 introduced support for an asynchronous gateway, which offers a convenient way to initiate flows when you may not know if a reply is expected or how long it takes for replies to arrive.

To handle these types of scenarios, Spring Integration uses `java.util.concurrent.Future` instances to support an asynchronous gateway.

From the XML configuration, nothing changes, and you still define asynchronous gateway the same way as you define a regular gateway, as the following example shows:

```
<int:gateway id="mathService"
     service-interface=
"org.springframework.integration.sample.gateway.futures.MathServiceGateway"
     default-request-channel="requestChannel"/>
```

However, the gateway interface (a service interface) is a little different, as follows:

```
public interface MathServiceGateway {

  Future<Integer> multiplyByTwo(int i);

}
```

As the preceding example shows, the return type for the gateway method is a `Future`. When `GatewayProxyFactoryBean` sees that the return type of the gateway method is a `Future`, it immediately switches to the asynchronous mode by using an `AsyncTaskExecutor`. That is the extent of the differences. The call to such a method always returns immediately with a `Future` instance. Then you can interact with the `Future` at your own pace to get the result, cancel, and so on. Also, as with any other use of `Future` instances, calling `get()` may reveal a timeout, an execution exception, and so on. The following example shows how to use a `Future` that returns from an asynchronous gateway:

```
MathServiceGateway mathService = ac.getBean("mathService", MathServiceGateway
.class);
Future<Integer> result = mathService.multiplyByTwo(number);
// do something else here since the reply might take a moment
int finalResult =  result.get(1000, TimeUnit.SECONDS);
```

For a more detailed example, see the async-gateway sample in the Spring Integration samples.

`ListenableFuture`

Starting with version 4.1, asynchronous gateway methods can also return `ListenableFuture` (introduced in Spring Framework 4.0). These return types let you provide a callback, which is invoked when the result is available (or an exception occurs). When the gateway detects this return type and the task executor is an `AsyncListenableTaskExecutor`, the executor's `submitListenable()` method is invoked. The following example shows how to use a `ListenableFuture`:

```
ListenableFuture<String> result = this.asyncGateway.async("something");
result.addCallback(new ListenableFutureCallback<String>() {

    @Override
    public void onSuccess(String result) {
        ...
    }

    @Override
    public void onFailure(Throwable t) {
        ...
    }
});
```

AsyncTaskExecutor

By default, the GatewayProxyFactoryBean uses org.springframework.core.task.SimpleAsyncTaskExecutor when submitting internal AsyncInvocationTask instances for any gateway method whose return type is a Future. However, the async-executor attribute in the <gateway/> element's configuration lets you provide a reference to any implementation of java.util.concurrent.Executor available within the Spring application context.

The (default) SimpleAsyncTaskExecutor supports both Future and ListenableFuture return types, returning FutureTask or ListenableFutureTask respectively. See CompletableFuture. Even though there is a default executor, it is often useful to provide an external one so that you can identify its threads in logs (when using XML, the thread name is based on the executor's bean name), as the following example shows:

```
@Bean
public AsyncTaskExecutor exec() {
    SimpleAsyncTaskExecutor simpleAsyncTaskExecutor = new SimpleAsyncTaskExecutor
();
    simpleAsyncTaskExecutor.setThreadNamePrefix("exec-");
    return simpleAsyncTaskExecutor;
}

@MessagingGateway(asyncExecutor = "exec")
public interface ExecGateway {

    @Gateway(requestChannel = "gatewayChannel")
    Future<?> doAsync(String foo);

}
```

If you wish to return a different `Future` implementation, you can provide a custom executor or disable the executor altogether and return the `Future` in the reply message payload from the downstream flow. To disable the executor, set it to `null` in the `GatewayProxyFactoryBean` (by using `setAsyncTaskExecutor(null)`). When configuring the gateway with XML, use `async-executor=""`. When configuring by using the `@MessagingGateway` annotation, use code similar to the following:

```
@MessagingGateway(asyncExecutor = AnnotationConstants.NULL)
public interface NoExecGateway {

    @Gateway(requestChannel = "gatewayChannel")
    Future<?> doAsync(String foo);

}
```

If the return type is a specific concrete `Future` implementation or some other sub-interface that is not supported by the configured executor, the flow runs on the caller's thread and the flow must return the required type in the reply message payload.

`CompletableFuture`

Starting with version 4.2, gateway methods can now return `CompletableFuture<?>`. There are two modes of operation when returning this type:

- When an async executor is provided and the return type is exactly `CompletableFuture` (not a subclass), the framework runs the task on the executor and immediately returns a `CompletableFuture` to the caller. `CompletableFuture.supplyAsync(Supplier<U> supplier, Executor executor)` is used to create the future.
- When the async executor is explicitly set to `null` and the return type is `CompletableFuture` or the return type is a subclass of `CompletableFuture`, the flow is invoked on the caller's thread. In this scenario, the downstream flow is expected to return a `CompletableFuture` of the appropriate type.

Usage Scenarios

In the following scenario, the caller thread returns immediately with a `CompletableFuture<Invoice>`, which is completed when the downstream flow replies to the gateway (with an `Invoice` object).

```
CompletableFuture<Invoice> order(Order order);
```

```
<int:gateway service-interface="something.Service" default-request-channel="
orders" />
```

In the following scenario, the caller thread returns with a `CompletableFuture<Invoice>` when the downstream flow provides it as the payload of the reply to the gateway. Some other process must complete the future when the invoice is ready.

```
CompletableFuture<Invoice> order(Order order);
```

```
<int:gateway service-interface="foo.Service" default-request-channel="orders"
    async-executor="" />
```

In the following scenario, the caller thread returns with a `CompletableFuture<Invoice>` when the downstream flow provides it as the payload of the reply to the gateway. Some other process must complete the future when the invoice is ready. If `DEBUG` logging is enabled, a log entry is emitted, indicating that the async executor cannot be used for this scenario.

```
MyCompletableFuture<Invoice> order(Order order);
```

```
<int:gateway service-interface="foo.Service" default-request-channel="orders" />
```

`CompletableFuture` instances can be used to perform additional manipulation on the reply, as the following example shows:

```
CompletableFuture<String> process(String data);

...

CompletableFuture result = process("foo")
    .thenApply(t -> t.toUpperCase());

...

String out = result.get(10, TimeUnit.SECONDS);
```

Reactor `Mono`

Starting with version 5.0, the `GatewayProxyFactoryBean` allows the use of Project Reactor with gateway interface methods, using a `Mono<T>` return type. The internal `AsyncInvocationTask` is wrapped in a `Mono.fromCallable()`.

A `Mono` can be used to retrieve the result later (similar to a `Future<?>`), or you can consume from it

with the dispatcher by invoking your `Consumer` when the result is returned to the gateway.

The `Mono` is not immediately flushed by the framework. Consequently, the underlying message flow is not started before the gateway method returns (as it is with a `Future<?> Executor` task). The flow starts when the `Mono` is subscribed to. Alternatively, the `Mono` (being a "Composable") might be a part of Reactor stream, when the `subscribe()` is related to the entire `Flux`. The following example shows how to create a gateway with Project Reactor:

```
@MessagingGateway
public static interface TestGateway {

    @Gateway(requestChannel = "promiseChannel")
    Mono<Integer> multiply(Integer value);

    }

        ...

    @ServiceActivator(inputChannel = "promiseChannel")
    public Integer multiply(Integer value) {
            return value * 2;
    }

        ...

    Flux.just("1", "2", "3", "4", "5")
            .map(Integer::parseInt)
            .flatMap(this.testGateway::multiply)
            .collectList()
            .subscribe(integers -> ...);
```

Another example that uses Project Reactor is a simple callback scenario, as the following example shows:

```
Mono<Invoice> mono = service.process(myOrder);

mono.subscribe(invoice -> handleInvoice(invoice));
```

The calling thread continues, with `handleInvoice()` being called when the flow completes.

Downstream Flows Returning an Asynchronous Type

As mentioned in the `ListenableFuture` section above, if you wish some downstream component to

return a message with an async payload (`Future`, `Mono`, and others), you must explicitly set the async executor to `null` (or `""` when using XML configuration). The flow is then invoked on the caller thread and the result can be retrieved later.

`void` Return Type

Unlike the return types mentioned earlier, when the method return type is `void`, the framework cannot implicitly determine that you wish the downstream flow to run asynchronously, with the caller thread returning immediately. In this case, you must annotate the interface method with `@Async`, as the following example shows:

```
@MessagingGateway
public interface MyGateway {

    @Gateway(requestChannel = "sendAsyncChannel")
    @Async
    void sendAsync(String payload);

}
```

Unlike the `Future<?>` return types, there is no way to inform the caller if some exception is thrown by the flow, unless some custom `TaskExecutor` (such as an `ErrorHandlingTaskExecutor`) is associated with the `@Async` annotation.

10.4.12. Gateway Behavior When No response Arrives

As explained earlier, the gateway provides a convenient way of interacting with a messaging system through POJO method invocations. However, a typical method invocation, which is generally expected to always return (even with an Exception), might not always map one-to-one to message exchanges (for example, a reply message might not arrive — the equivalent to a method not returning).

The rest of this section covers various scenarios and how to make the gateway behave more predictably. Certain attributes can be configured to make synchronous gateway behavior more predictable, but some of them might not always work as you might expect. One of them is `reply-timeout` (at the method level or `default-reply-timeout` at the gateway level). We examine the `reply-timeout` attribute to see how it can and cannot influence the behavior of the synchronous gateway in various scenarios. We examine a single-threaded scenario (all components downstream are connected through a direct channel) and multi-threaded scenarios (for example, somewhere downstream you may have a pollable or executor channel that breaks the single-thread boundary).

Long-running Process Downstream

Sync Gateway, single-threaded

If a component downstream is still running (perhaps because of an infinite loop or a slow service), setting a `reply-timeout` has no effect, and the gateway method call does not return until the downstream service exits (by returning or throwing an exception).

Sync Gateway, multi-threaded

If a component downstream is still running (perhaps because of an infinite loop or a slow service) in a multi-threaded message flow, setting the `reply-timeout` has an effect by allowing gateway method invocation to return once the timeout has been reached, because the `GatewayProxyFactoryBean` polls on the reply channel, waiting for a message until the timeout expires. However, if the timeout has been reached before the actual reply was produced, it could result in a 'null' return from the gateway method. You should understand that the reply message (if produced) is sent to a reply channel after the gateway method invocation might have returned, so you must be aware of that and design your flow with it in mind.

Downstream Component Returns 'null'

Sync Gateway — single-threaded

If a component downstream returns 'null' and no `reply-timeout` has been configured, the gateway method call hangs indefinitely, unless a `reply-timeout` has been configured or the `requires-reply` attribute has been set on the downstream component (for example, a service activator) that might return 'null'. In this case, an exception would be thrown and propagated to the gateway.

Sync Gateway — multi-threaded

The behavior is the same as the previous case.

Downstream Component Return Signature is 'void' While Gateway Method Signature Is Non-void

Sync Gateway — single-threaded

If a component downstream returns 'void' and no `reply-timeout` has been configured, the gateway method call hangs indefinitely unless a `reply-timeout` has been configured.

Sync Gateway — multi-threaded

The behavior is the same as the previous case.

Downstream Component Results in Runtime Exception

Sync Gateway — single-threaded

If a component downstream throws a runtime exception, the exception is propagated through an error message back to the gateway and re-thrown.

Sync Gateway — multi-threaded

The behavior is the same as the previous case.

You should understand that, by default, `reply-timeout` is unbounded. Consequently, if you do not explicitly set the `reply-timeout`, your gateway method invocation might hang indefinitely. So, to make sure you analyze your flow and if there is even a remote possibility of one of these scenarios to occur, you should set the `reply-timeout` attribute to a "'safe'" value. Even better, you can set the `requires-reply` attribute of the downstream component to 'true' to ensure a timely response, as produced by the throwing of an exception as soon as that downstream component returns null internally. However you should also realize that there are some scenarios (see the first one) where `reply-timeout` does not help. That means it is also important to analyze your message flow and decide when to use a synchronous gateway rather than an asynchrnous gateway. As described earlier, the latter case is a matter of defining gateway methods that return `Future` instances. Then you are guaranteed to receive that return value, and you have more granular control over the results of the invocation. Also, when dealing with a router, you should remember that setting the `resolution-required` attribute to 'true' results in an exception thrown by the router if it can not resolve a particular channel. Likewise, when dealing with a Filter, you can set the `throw-exception-on-rejection` attribute. In both of these cases, the resulting flow behaves like it contain a service activator with the 'requires-reply' attribute. In other words, it helps to ensure a timely response from the gateway method invocation.

`reply-timeout` is unbounded for `<gateway/>` elements (created by the `GatewayProxyFactoryBean`). Inbound gateways for external integration (WS, HTTP, and so on) share many characteristics and attributes with these gateways. However, for those inbound gateways, the default `reply-timeout` is 1000 milliseconds (one second). If a downstream asynchronous hand-off is made to another thread, you may need to increase this attribute to allow enough time for the flow to complete before the gateway times out.

You should understand that the timer starts when the thread returns to the gateway — that is, when the flow completes or a message is handed off to another thread. At that time, the calling thread starts waiting for the reply. If the flow was completely synchronous, the reply is immediately available. For asynchronous flows, the thread waits for up to this time.

See `IntegrationFlow` as Gateway in the Java DSL chapter for options to define gateways through `IntegrationFlows`.

10.5. Service Activator

The service activator is the endpoint type for connecting any Spring-managed object to an input channel so that it may play the role of a service. If the service produces output, it may also be connected to an output channel. Alternatively, an output-producing service may be located at the end of a processing pipeline or message flow, in which case the inbound message's `replyChannel` header can be used. This is the default behavior if no output channel is defined. As with most of the configuration options described here, the same behavior actually applies for most of the other

components.

10.5.1. Configuring Service Activator

To create a service activator, use the 'service-activator' element with the 'input-channel' and 'ref' attributes, as the following example shows:

```
<int:service-activator input-channel="exampleChannel" ref="exampleHandler"/>
```

The preceding configuration selects all the methods from the `exampleHandler` that meet one of the messaging requirements, which are as follows:

- annotated with `@ServiceActivator`
- is `public`
- not return `void` if `requiresReply == true`

The target method for invocation at runtime is selected for each request message by their `payload` type or as a fallback to the `Message<?>` type if such a method is present on target class.

Starting with version 5.0, one service method can be marked with the `@org.springframework.integration.annotation.Default` as a fallback for all non-matching cases. This can be useful when using content-type conversion with the target method being invoked after conversion.

To delegate to an explicitly defined method of any object, you can add the `method` attribute, as the following example shows:

```
<int:service-activator input-channel="exampleChannel" ref="somePojo" method=
"someMethod"/>
```

In either case, when the service method returns a non-null value, the endpoint tries to send the reply message to an appropriate reply channel. To determine the reply channel, it first checks whether an `output-channel` was provided in the endpoint configuration, as the following example shows:

```
<int:service-activator input-channel="exampleChannel" output-channel="
replyChannel"
                       ref="somePojo" method="someMethod"/>
```

If the method returns a result and no `output-channel` is defined, the framework then checks the request message's `replyChannel` header value. If that value is available, it then checks its type. If it is

a `MessageChannel`, the reply message is sent to that channel. If it is a `String`, the endpoint tries to resolve the channel name to a channel instance. If the channel cannot be resolved, a `DestinationResolutionException` is thrown. It it can be resolved, the message is sent there. If the request message does not have a `replyChannel` header and the `reply` object is a `Message`, its `replyChannel` header is consulted for a target destination. This is the technique used for request-reply messaging in Spring Integration, and it is also an example of the return address pattern.

If your method returns a result and you want to discard it and end the flow, you should configure the `output-channel` to send to a `NullChannel`. For convenience, the framework registers one with the name, `nullChannel`. See Special Channels for more information.

The service activator is one of those components that is not required to produce a reply message. If your method returns `null` or has a `void` return type, the service activator exits after the method invocation, without any signals. This behavior can be controlled by the `AbstractReplyProducingMessageHandler.requiresReply` option, which is also exposed as `requires-reply` when configuring with the XML namespace. If the flag is set to `true` and the method returns null, a `ReplyRequiredException` is thrown.

The argument in the service method could be either a message or an arbitrary type. If the latter, then it is assumed to be a message payload, which is extracted from the message and injected into the service method. We generally recommend this approach, as it follows and promotes a POJO model when working with Spring Integration. Arguments may also have `@Header` or `@Headers` annotations, as described in Annotation Support.

The service method is not required to have any arguments, which means you can implement event-style service activators (where all you care about is an invocation of the service method) and not worry about the contents of the message. Think of it as a null JMS message. An example use case for such an implementation is a simple counter or monitor of messages deposited on the input channel.

Starting with version 4.1, the framework correctly converts message properties (`payload` and `headers`) to the Java 8 `Optional` POJO method parameters, as the following example shows:

```
public class MyBean {
    public String computeValue(Optional<String> payload,
                @Header(value="foo", required=false) String foo1,
                @Header(value="foo") Optional<String> foo2) {
        if (payload.isPresent()) {
            String value = payload.get();
            ...
        }
        else {
           ...
       }
    }

}
```

We generally recommend using a `ref` attribute if the custom service activator handler implementation can be reused in other `<service-activator>` definitions. However, if the custom service activator handler implementation is only used within a single definition of the `<service-activator>`, you can provide an inner bean definition, as the following example shows:

```
<int:service-activator id="exampleServiceActivator" input-channel="inChannel"
            output-channel = "outChannel" method="someMethod">
    <beans:bean class="org.something.ExampleServiceActivator"/>
</int:service-activator>
```

Using both the `ref` attribute and an inner handler definition in the same `<service-activator>` configuration is not allowed, as it creates an ambiguous condition and results in an exception being thrown.

If the `ref` attribute references a bean that extends `AbstractMessageProducingHandler` (such as handlers provided by the framework itself), the configuration is optimized by injecting the output channel into the handler directly. In this case, each `ref` must be to a separate bean instance (or a `prototype`-scoped bean) or use the inner `<bean/>` configuration type. If you inadvertently reference the same message handler from multiple beans, you get a configuration exception.

Service Activators and the Spring Expression Language (SpEL)

Since Spring Integration 2.0, service activators can also benefit from SpEL.

For example, you can invoke any bean method without pointing to the bean in a `ref` attribute or including it as an inner bean definition, as follows:

```
<int:service-activator input-channel="in" output-channel="out"
    expression="@accountService.processAccount(payload, headers.accountId)"/>

    <bean id="accountService" class="thing1.thing2.Account"/>
```

In the preceding configuration, instead of injecting 'accountService' by using a `ref` or as an inner bean, we use SpEL's `@beanId` notation and invoke a method that takes a type compatible with the message payload. We also pass a header value. Any valid SpEL expression can be evaluated against any content in the message. For simple scenarios, your service activators need not reference a bean if all logic can be encapsulated in such an expression, as the following example shows:

```
<int:service-activator input-channel="in" output-channel="out" expression="payload
* 2"/>
```

In the preceding configuration, our service logic is to multiply the payload value by two. SpEL lets us handle it relatively easily.

See Service Activators and the `.handle()` method in the Java DSL chapter for more information about configuring service activator.

10.5.2. Asynchronous Service Activator

The service activator is invoked by the calling thread. This is an upstream thread if the input channel is a `SubscribableChannel` or a poller thread for a `PollableChannel`. If the service returns a `ListenableFuture<?>`, the default action is to send that as the payload of the message sent to the output (or reply) channel. Starting with version 4.3, you can now set the `async` attribute to `true` (by using `setAsync(true)` when using Java configuration). If the service returns a `ListenableFuture<?>` when this the `async` attribute is set to `true`, the calling thread is released immediately and the reply message is sent on the thread (from within your service) that completes the future. This is particularly advantageous for long-running services that use a `PollableChannel`, because the poller thread is released to perform other services within the framework.

If the service completes the future with an `Exception`, normal error processing occurs. An `ErrorMessage` is sent to the `errorChannel` message header, if present. Otherwise, an `ErrorMessage` is sent to the default `errorChannel` (if available).

10.5.3. Service Activator and Method Return Type

The service method can return any type which becomes reply message payload. In this case a new `Message<?>` object is created and all the headers from a request message are copied. This works the same way for most Spring Integration `MessageHandler` implementations, when interaction is based on a POJO method invocation.

A complete `Message<?>` object can also be returned from the method. However keep in mind that,

unlike transformers, for a Service Activator this message will be modified by copying the headers from the request message if they are not already present in the returned message. So, if your method parameter is a `Message<?>` and you copy some, but not all, existing headers in your service method, they will reappear in the reply message. It is not a Service Activator responsibility to remove headers from a reply message and, pursuing the loosely-coupled principle, it is better to add a `HeaderFilter` in the integration flow. Alternatively, a Transformer can be used instead of a Service Activator but, in that case, when returning a full `Message<?>` the method is completely responsible for the message, including copying request message headers (if needed). You must ensure that important framework headers (e.g. `replyChannel`, `errorChannel`), if present, have to be preserved.

10.6. Delayer

A delayer is a simple endpoint that lets a message flow be delayed by a certain interval. When a message is delayed, the original sender does not block. Instead, the delayed messages are scheduled with an instance of `org.springframework.scheduling.TaskScheduler` to be sent to the output channel after the delay has passed. This approach is scalable even for rather long delays, since it does not result in a large number of blocked sender threads. On the contrary, in the typical case, a thread pool is used for the actual execution of releasing the messages. This section contains several examples of configuring a delayer.

10.6.1. Configuring a Delayer

The `<delayer>` element is used to delay the message flow between two message channels. As with the other endpoints, you can provide the 'input-channel' and 'output-channel' attributes, but the delayer also has 'default-delay' and 'expression' attributes (and the 'expression' element) that determine the number of milliseconds by which each message should be delayed. The following example delays all messages by three seconds:

```
<int:delayer id="delayer" input-channel="input"
             default-delay="3000" output-channel="output"/>
```

If you need to determine the delay for each message, you can also provide the SpEL expression by using the 'expression' attribute, as the following expression shows:

```
<int:delayer id="delayer" input-channel="input" output-channel="output"
             default-delay="3000" expression="headers['delay']"/>
```

In the preceding example, the three-second delay applies only when the expression evaluates to null for a given inbound message. If you want to apply a delay only to messages that have a valid result of the expression evaluation, you can use a 'default-delay' of `0` (the default). For any message that has a delay of `0` (or less), the message is sent immediately, on the calling thread.

The following example shows the Java configuration equivalent of the preceding example:

```
@ServiceActivator(inputChannel = "input")
@Bean
public DelayHandler delayer() {
    DelayHandler handler = new DelayHandler("delayer.messageGroupId");
    handler.setDefaultDelay(3_000L);
    handler.setDelayExpressionString("headers['delay']");
    handler.setOutputChannelName("output");
    return handler;
}
```

The following example shows the Java DSL equivalent of the preceding example:

```
@Bean
public IntegrationFlow flow() {
    return IntegrationFlows.from("input")
            .delay("delayer.messageGroupId", d -> d
                    .defaultDelay(3_000L)
                    .delayExpression("headers['delay']"))
            .channel("output")
            .get();
}
```

The XML parser uses a message group ID of `<beanName>.messageGroupId`.

The delay handler supports expression evaluation results that represent an interval in milliseconds (any `Object` whose `toString()` method produces a value that can be parsed into a `Long`) as well as `java.util.Date` instances representing an absolute time. In the first case, the milliseconds are counted from the current time (for example a value of `5000` would delay the message for at least five seconds from the time it is received by the delayer). With a `Date` instance, the message is not released until the time represented by that `Date` object. A value that equates to a non-positive delay or a Date in the past results in no delay. Instead, it is sent directly to the output channel on the original sender's thread. If the expression evaluation result is not a `Date` and can not be parsed as a `Long`, the default delay (if any — the default is `0`) is applied.

The expression evaluation may throw an evaluation exception for various reasons, including an invalid expression or other conditions. By default, such exceptions are ignored (though logged at the DEBUG level) and the delayer falls back to the default delay (if any). You can modify this behavior by setting the `ignore-expression-failures` attribute. By default, this attribute is set to `true` and the delayer behavior is as described earlier. However, if you wish to not ignore expression evaluation exceptions and throw them to the delayer's caller, set the `ignore-expression-failures` attribute to `false`.

In the preceding example, the delay expression is specified as `headers['delay']`. This is the SpEL `Indexer` syntax to access a `Map` element (`MessageHeaders` implements `Map`). It invokes: `headers.get("delay")`. For simple map element names (that do not contain '.') you can also use the SpEL "dot accessor" syntax, where the header expression shown earlier can be specified as `headers.delay`. However, different results are achieved if the header is missing. In the first case, the expression evaluates to `null`. The second results in something similar to the following:

```
 org.springframework.expression.spel.SpelEvaluationException:
EL1008E:(pos 8):
           Field or property 'delay' cannot be found on object of
type 'org.springframework.messaging.MessageHeaders'
```

Consequently, if there is a possibility of the header being omitted and you want to fall back to the default delay, it is generally more efficient (and recommended) to use the indexer syntax instead of dot property accessor syntax, because detecting the null is faster than catching an exception.

The delayer delegates to an instance of Spring's `TaskScheduler` abstraction. The default scheduler used by the delayer is the `ThreadPoolTaskScheduler` instance provided by Spring Integration on startup. See Configuring the Task Scheduler. If you want to delegate to a different scheduler, you can provide a reference through the delayer element's 'scheduler' attribute, as the following example shows:

```
<int:delayer id="delayer" input-channel="input" output-channel="output"
    expression="headers.delay"
    scheduler="exampleTaskScheduler"/>

<task:scheduler id="exampleTaskScheduler" pool-size="3"/>
```

If you configure an external `ThreadPoolTaskScheduler`, you can set `waitForTasksToCompleteOnShutdown = true` on this property. It allows successful completion of 'delay' tasks that are already in the execution state (releasing the message) when the application is shutdown. Before Spring Integration 2.2, this property was available on the `<delayer>` element, because `DelayHandler` could create its own scheduler on the background. Since 2.2, the delayer requires an external scheduler instance and `waitForTasksToCompleteOnShutdown` was deleted. You should use the scheduler's own configuration.

`ThreadPoolTaskScheduler` has a property `errorHandler`, which can be injected with some implementation of `org.springframework.util.ErrorHandler`. This handler allows processing an `Exception` from the thread of the scheduled task sending the delayed message. By default, it uses an `org.springframework.scheduling.support.TaskUtils$LoggingErrorHandler`, and you can see a stack trace in the logs. You might want to consider using an `org.springframework.integration.channel.MessagePublishingErrorHandler`, which sends an `ErrorMessage` into an `error-channel`, either from the failed message's header or into the default `error-channel`. This error handling is performed after a transaction rolls back (if present). See Release Failures.

10.6.2. Delayer and a Message Store

The `DelayHandler` persists delayed messages into the message group in the provided `MessageStore`. (The 'groupId' is based on the required 'id' attribute of the `<delayer>` element.) A delayed message is removed from the `MessageStore` by the scheduled task immediately before the `DelayHandler` sends the message to the `output-channel`. If the provided `MessageStore` is persistent (such as `JdbcMessageStore`), it provides the ability to not lose messages on the application shutdown. After application startup, the `DelayHandler` reads messages from its message group in the `MessageStore` and reschedules them with a delay based on the original arrival time of the message (if the delay is numeric). For messages where the delay header was a `Date`, that `Date` is used when rescheduling. If a delayed message remains in the `MessageStore` more than its 'delay', it is sent immediately after startup.

The `<delayer>` can be enriched with either of two mutually exclusive elements: `<transactional>` and `<advice-chain>`. The `List` of these AOP advices is applied to the proxied internal `DelayHandler.ReleaseMessageHandler`, which has the responsibility to release the message, after the delay, on a `Thread` of the scheduled task. It might be used, for example, when the downstream message flow throws an exception and the transaction of the `ReleaseMessageHandler` is rolled back. In this case, the delayed message remains in the persistent `MessageStore`. You can use any custom `org.aopalliance.aop.Advice` implementation within the `<advice-chain>`. The `<transactional>` element defines a simple advice chain that has only the transactional advice. The following example shows an `advice-chain` within a `<delayer>`:

```
<int:delayer id="delayer" input-channel="input" output-channel="output"
    expression="headers.delay"
    message-store="jdbcMessageStore">
    <int:advice-chain>
        <beans:ref bean="customAdviceBean"/>
        <tx:advice>
            <tx:attributes>
                <tx:method name="*" read-only="true"/>
            </tx:attributes>
        </tx:advice>
    </int:advice-chain>
</int:delayer>
```

The `DelayHandler` can be exported as a JMX `MBean` with managed operations (`getDelayedMessageCount` and `reschedulePersistedMessages`), which allows the rescheduling of delayed persisted messages at runtime — for example, if the `TaskScheduler` has previously been stopped. These operations can be invoked through a `Control Bus` command, as the following example shows:

```
Message<String> delayerReschedulingMessage =
    MessageBuilder.withPayload("@'delayer.handler'.reschedulePersistedMessages()"
).build();
    controlBusChannel.send(delayerReschedulingMessage);
```

For more information regarding the message store, JMX, and the control bus, see System Management.

10.6.3. Release Failures

Starting with version 5.0.8, there are two new properties on the delayer:

- `maxAttempts` (default 5)
- `retryDelay` (default 1 second)

When a message is released, if the downstream flow fails, the release will be attempted after the `retryDelay`. If the `maxAttempts` is reached, the message is discarded (unless the release is transactional, in which case the message will remain in the store, but will no longer be scheduled for release, until the application is restarted, or the `reschedulePersistedMessages()` method is invoked, as discussed above).

In addition, you can configure a `delayedMessageErrorChannel`; when a release fails, an `ErrorMessage` is sent to that channel with the exception as the payload and has the `originalMessage` property. The `ErrorMessage` contains a header `IntegrationMessageHeaderAccessor.DELIVERY_ATTEMPT` containing the current count.

If the error flow consumes the error message and exits normally, no further action is taken; if the release is transactional, the transaction will commit and the message deleted from the store. If the error flow throws an exception, the release will be retried up to `maxAttempts` as discussed above.

10.7. Scripting Support

Spring Integration 2.1 added support for the JSR223 Scripting for Java specification, introduced in Java version 6. It lets you use scripts written in any supported language (including Ruby, JRuby, Javascript, Groovy and Kotlin) to provide the logic for various integration components, similar to the way the Spring Expression Language (SpEL) is used in Spring Integration. For more information about JSR223, see the documentation.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-scripting</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-scripting:5.3.0.RC1"
```

In addition you need to add a script engine implementation, e.g. JRuby, Jython.

Starting with version 5.2, Spring Integration provides a Kotlin Jsr223 support. You need to add these dependencies into your project to make it working:

```
runtime 'org.jetbrains.kotlin:kotlin-script-util'
runtime 'org.jetbrains.kotlin:kotlin-compiler-embeddable'
runtime 'org.jetbrains.kotlin:kotlin-scripting-compiler-embeddable'
```

The `KotlinScriptExecutor` is selected by the provided `kotlin` language indicator or script file comes with the `.kts` extension.

Note that this feature requires Java 6 or higher.

In order to use a JVM scripting language, a JSR223 implementation for that language must be included in your class path. Java 6 natively supports Javascript. The Groovy and JRuby projects provide JSR233 support in their standard distributions.

Various JSR223 language implementations have been developed by third parties. A particular implementation's compatibility with Spring Integration depends on how well it conforms to the specification and the implementer's interpretation of the specification.

If you plan to use Groovy as your scripting language, we recommended you use Spring-Integration's Groovy Support as it offers additional features specific to Groovy. However, this section is relevant as well.

10.7.1. Script Configuration

Depending on the complexity of your integration requirements, scripts may be provided inline as CDATA in XML configuration or as a reference to a Spring resource that contains the script. To enable scripting support, Spring Integration defines a `ScriptExecutingMessageProcessor`, which binds the message payload to a variable named `payload` and the message headers to a `headers` variable, both accessible within the script execution context. All you need to do is write a script that uses these variables. The following pair of examples show sample configurations that create filters:

Example 2. Filter

```
<int:filter input-channel="referencedScriptInput">
   <int-script:script location="some/path/to/ruby/script/RubyFilterTests.rb"/>
</int:filter>

<int:filter input-channel="inlineScriptInput">
     <int-script:script lang="groovy">
     <![CDATA[
     return payload == 'good'
   ]]>
  </int-script:script>
</int:filter>
```

As the preceding examples show, the script can be included inline or can be included by reference to a resource location (by using the `location` attribute). Additionally, the `lang` attribute corresponds to the language name (or its JSR223 alias)

Other Spring Integration endpoint elements that support scripting include `router`, `service-activator`, `transformer`, and `splitter`. The scripting configuration in each case would be identical to the above (besides the endpoint element).

Another useful feature of scripting support is the ability to update (reload) scripts without having to restart the application context. To do so, specify the `refresh-check-delay` attribute on the `script` element, as the following example shows:

```
<int-script:script location="..." refresh-check-delay="5000"/>
```

In the preceding example, the script location is checked for updates every 5 seconds. If the script is updated, any invocation that occurs later than 5 seconds since the update results in running the new script.

Consider the following example:

```
<int-script:script location="..." refresh-check-delay="0"/>
```

In the preceding example, the context is updated with any script modifications as soon as such modification occurs, providing a simple mechanism for 'real-time' configuration. Any negative value means the script is not reloaded after initialization of the application context. This is the default behavior. The following example shows a script that never updates:

```
<int-script:script location="..." refresh-check-delay="-1"/>
```

Inline scripts can not be reloaded.

Script Variable Bindings

Variable bindings are required to enable the script to reference variables externally provided to the script's execution context. By default, `payload` and `headers` are used as binding variables. You can bind additional variables to a script by using `<variable>` elements, as the following example shows:

```
<script:script lang="js" location="foo/bar/MyScript.js">
    <script:variable name="foo" value="thing1"/>
    <script:variable name="bar" value="thing2"/>
    <script:variable name="date" ref="date"/>
</script:script>
```

As shown in the preceding example, you can bind a script variable either to a scalar value or to a Spring bean reference. Note that `payload` and `headers` are still included as binding variables.

With Spring Integration 3.0, in addition to the `variable` element, the `variables` attribute has been introduced. This attribute and the `variable` elements are not mutually exclusive, and you can combine them within one `script` component. However, variables must be unique, regardless of where they are defined. Also, since Spring Integration 3.0, variable bindings are allowed for inline

scripts, too, as the following example shows:

```
<service-activator input-channel="input">
    <script:script lang="ruby" variables="thing1=THING1, date-ref=dateBean">
        <script:variable name="thing2" ref="thing2Bean"/>
        <script:variable name="thing3" value="thing2"/>
        <![CDATA[
            payload.foo = thing1
            payload.date = date
            payload.bar = thing2
            payload.baz = thing3
            payload
        ]]>
    </script:script>
</service-activator>
```

The preceding example shows a combination of an inline script, a `variable` element, and a `variables` attribute. The `variables` attribute contains a comma-separated value, where each segment contains an '=' separated pair of the variable and its value. The variable name can be suffixed with `-ref`, as in the `date-ref` variable in the preceding example. That means that the binding variable has the name, `date`, but the value is a reference to the `dateBean` bean from the application context. This may be useful when using property placeholder configuration or command-line arguments.

If you need more control over how variables are generated, you can implement your own Java class that uses the `ScriptVariableGenerator` strategy, which is defined by the following interface:

```
public interface ScriptVariableGenerator {

    Map<String, Object> generateScriptVariables(Message<?> message);

}
```

This interface requires you to implement the `generateScriptVariables(Message)` method. The message argument lets you access any data available in the message payload and headers, and the return value is the `Map` of bound variables. This method is called every time the script is executed for a message. The following example shows how to provide an implementation of `ScriptVariableGenerator` and reference it with the `script-variable-generator` attribute:

```
<int-script:script location="foo/bar/MyScript.groovy"
        script-variable-generator="variableGenerator"/>

<bean id="variableGenerator" class="foo.bar.MyScriptVariableGenerator"/>
```

If a `script-variable-generator` is not provided, script components use `DefaultScriptVariableGenerator`, which merges any provided `<variable>` elements with `payload` and `headers` variables from the `Message` in its `generateScriptVariables(Message)` method.

You cannot provide both the `script-variable-generator` attribute and `<variable>` element(s). They are mutually exclusive.

10.8. Groovy support

In Spring Integration 2.0, we added Groovy support, letting you use the Groovy scripting language to provide the logic for various integration components — similar to the way the Spring Expression Language (SpEL) is supported for routing, transformation, and other integration concerns. For more information about Groovy, see the Groovy documentation, which you can find on the project website.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-groovy</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-groovy:5.3.0.RC1"
```

10.8.1. Groovy Configuration

With Spring Integration 2.1, the configuration namespace for the Groovy support is an extension of Spring Integration's scripting support and shares the core configuration and behavior described in detail in the Scripting Support section. Even though Groovy scripts are well supported by generic scripting support, the Groovy support provides the `Groovy` configuration namespace, which is backed by the Spring Framework's `org.springframework.scripting.groovy.GroovyScriptFactory` and related components, offering extended capabilities for using Groovy. The following listing shows two sample configurations:

Example 3. Filter

```
<int:filter input-channel="referencedScriptInput">
   <int-groovy:script location="some/path/to/groovy/file/GroovyFilterTests.groovy
"/>
</int:filter>

<int:filter input-channel="inlineScriptInput">
     <int-groovy:script><![CDATA[
     return payload == 'good'
   ]]></int-groovy:script>
</int:filter>
```

As the preceding examples show, the configuration looks identical to the general scripting support configuration. The only difference is the use of the Groovy namespace, as indicated by the `int-groovy` namespace prefix. Also note that the `lang` attribute on the `<script>` tag is not valid in this namespace.

10.8.2. Groovy Object Customization

If you need to customize the Groovy object itself (beyond setting variables) you can reference a bean that implements `GroovyObjectCustomizer` by using the `customizer` attribute. For example, this might be useful if you want to implement a domain-specific language (DSL) by modifying the `MetaClass` and registering functions to be available within the script. The following example shows how to do so:

```
<int:service-activator input-channel="groovyChannel">
    <int-groovy:script location="somewhere/SomeScript.groovy" customizer=
"groovyCustomizer"/>
</int:service-activator>

<beans:bean id="groovyCustomizer" class="org.something.MyGroovyObjectCustomizer"/>
```

Setting a custom `GroovyObjectCustomizer` is not mutually exclusive with `<variable>` elements or the `script-variable-generator` attribute. It can also be provided when defining an inline script.

Spring Integration 3.0 introduced the `variables` attribute, which works in conjunction with the `variable` element. Also, groovy scripts have the ability to resolve a variable to a bean in the `BeanFactory`, if a binding variable was not provided with the name. The following example shows how to use a variable (`entityManager`):

```
<int-groovy:script>
    <![CDATA[
        entityManager.persist(payload)
        payload
    ]]>
</int-groovy:script>
```

`entityManager` must be an appropriate bean in the application context.

For more information regarding the `<variable>` element, the `variables` attribute, and the `script-variable-generator` attribute, see Script Variable Bindings.

10.8.3. Groovy Script Compiler Customization

The `@CompileStatic` hint is the most popular Groovy compiler customization option. It can be used on the class or method level. For more information, see the Groovy Reference Manual and, specifically, @CompileStatic. To utilize this feature for short scripts (in integration scenarios), we are forced to change simple scripts to more Java-like code. Consider the following `<filter>` script:

```
headers.type == 'good'
```

The preceding script becomes the following method in Spring Integration:

```
@groovy.transform.CompileStatic
String filter(Map headers) {
    headers.type == 'good'
}

filter(headers)
```

With that, the `filter()` method is transformed and compiled to static Java code, bypassing the Groovy dynamic phases of invocation, such as `getProperty()` factories and `CallSite` proxies.

Starting with version 4.3, you can configure the Spring Integration Groovy components with the `compile-static boolean` option, specifying that `ASTTransformationCustomizer` for `@CompileStatic` should be added to the internal `CompilerConfiguration`. With that in place, you can omit the method declaration with `@CompileStatic` in our script code and still get compiled plain Java code. In this case, the preceding script can be short but still needs to be a little more verbose than interpreted script, as the following example shows:

```
binding.variables.headers.type == 'good'
```

You must access the `headers` and `payload` (or any other) variables through the `groovy.lang.Script binding` property because, with `@CompileStatic`, we do not have the dynamic `GroovyObject.getProperty()` capability.

In addition, we introduced the `compiler-configuration` bean reference. With this attribute, you can provide any other required Groovy compiler customizations, such as `ImportCustomizer`. For more information about this feature, see the Groovy Documentation for advanced compiler configuration.

Using `compilerConfiguration` does not automatically add an `ASTTransformationCustomizer` for the `@CompileStatic` annotation, and it overrides the `compileStatic` option. If you still need `CompileStatic`, you should manually add a `new ASTTransformationCustomizer(CompileStatic.class)` into the `CompilationCustomizers` of that custom `compilerConfiguration`.

The Groovy compiler customization does not have any effect on the `refresh-check-delay` option, and reloadable scripts can be statically compiled, too.

10.8.4. Control Bus

As described in (Enterprise Integration Patterns), the idea behind the control bus is that you can use the same messaging system for monitoring and managing the components within the framework as is used for "application-level" messaging. In Spring Integration, we build upon the adapters described earlier so that you can send Messages as a means of invoking exposed operations. One option for those operations is Groovy scripts. The following example configures a Groovy script for the control bus:

```
<int-groovy:control-bus input-channel="operationChannel"/>
```

The control bus has an input channel that can be accessed to invoke operations on the beans in the application context.

The Groovy control bus runs messages on the input channel as Groovy scripts. It takes a message, compiles the body to a script, customizes it with a `GroovyObjectCustomizer`, and runs it. The control bus' `MessageProcessor` exposes all beans in the application context that are annotated with `@ManagedResource` and implement Spring's `Lifecycle` interface or extend Spring's `CustomizableThreadCreator` base class (for example, several of the `TaskExecutor` and `TaskScheduler` implementations).

Be careful about using managed beans with custom scopes (such as 'request') in the Control Bus' command scripts, especially inside an asynchronous message flow. If `MessageProcessor` of the control bus cannot expose a bean from the application context, you may end up with some `BeansException` during the command script's run. For example, if a custom scope's context is not established, the attempt to get a bean within that scope triggers a `BeanCreationException`.

If you need to further customize the Groovy objects, you can also provide a reference to a bean that implements `GroovyObjectCustomizer` through the `customizer` attribute, as the following example shows:

```
<int-groovy:control-bus input-channel="input"
        output-channel="output"
        customizer="groovyCustomizer"/>

<beans:bean id="groovyCustomizer" class="org.foo.MyGroovyObjectCustomizer"/>
```

10.9. Adding Behavior to Endpoints

Prior to Spring Integration 2.2, you could add behavior to an entire Integration flow by adding an AOP Advice to a poller's `<advice-chain/>` element. However, suppose you want to retry, say, just a REST Web Service call, and not any downstream endpoints.

For example, consider the following flow:

```
inbound-adapter->poller->http-gateway1->http-gateway2->jdbc-outbound-adapter
```

If you configure some retry-logic into an advice chain on the poller and the call to `http-gateway2` failed because of a network glitch, the retry causes both `http-gateway1` and `http-gateway2` to be called a second time. Similarly, after a transient failure in the jdbc-outbound-adapter, both HTTP gateways are called a second time before again calling the `jdbc-outbound-adapter`.

Spring Integration 2.2 adds the ability to add behavior to individual endpoints. This is achieved by the addition of the `<request-handler-advice-chain/>` element to many endpoints. The following example shows how to the `<request-handler-advice-chain/>` element within an `outbound-gateway`:

```
<int-http:outbound-gateway id="withAdvice"
    url-expression="'http://localhost/test1'"
    request-channel="requests"
    reply-channel="nextChannel">
    <int:request-handler-advice-chain>
        <ref bean="myRetryAdvice" />
    </request-handler-advice-chain>
</int-http:outbound-gateway>
```

In this case, `myRetryAdvice` is applied only locally to this gateway and does not apply to further actions taken downstream after the reply is sent to `nextChannel`. The scope of the advice is limited to the endpoint itself.

At this time, you cannot advise an entire `<chain/>` of endpoints. The schema does not allow a `<request-handler-advice-chain>` as a child element of the chain itself.

However, a `<request-handler-advice-chain>` can be added to individual reply-producing endpoints within a `<chain>` element. An exception is that, in a chain that produces no reply, because the last element in the chain is an `outbound-channel-adapter`, that last element cannot be advised. If you need to advise such an element, it must be moved outside of the chain (with the `output-channel` of the chain being the `input-channel` of the adapter). The adapter can then be advised as usual. For chains that produce a reply, every child element can be advised.

10.9.1. Provided Advice Classes

In addition to providing the general mechanism to apply AOP advice classes, Spring Integration provides these out-of-the-box advice implementations:

- `RequestHandlerRetryAdvice` (described in Retry Advice)
- `RequestHandlerCircuitBreakerAdvice` (described in Circuit Breaker Advice)
- `ExpressionEvaluatingRequestHandlerAdvice` (described in Expression Evaluating Advice)
- `RateLimiterRequestHandlerAdvice` (described in Rate Limiter Advice)
- `CacheRequestHandlerAdvice` (described in Caching Advice)
- `ReactiveRequestHandlerAdvice` (described in Reactive Advice)

Retry Advice

The retry advice (`o.s.i.handler.advice.RequestHandlerRetryAdvice`) leverages the rich retry mechanisms provided by the Spring Retry project. The core component of `spring-retry` is the `RetryTemplate`, which allows configuration of sophisticated retry scenarios, including `RetryPolicy` and `BackoffPolicy` strategies (with a number of implementations) as well as a `RecoveryCallback` strategy to determine the action to take when retries are exhausted.

Stateless Retry

Stateless retry is the case where the retry activity is handled entirely within the advice. The thread pauses (if configured to do so) and retries the action.

Stateful Retry

Stateful retry is the case where the retry state is managed within the advice but where an exception is thrown and the caller resubmits the request. An example for stateful retry is when we want the message originator (for example,JMS) to be responsible for resubmitting, rather than performing it on the current thread. Stateful retry needs some mechanism to detect a retried submission.

For more information on `spring-retry`, see the project's Javadoc and the reference documentation

for Spring Batch, where spring-retry originated.

The default back off behavior is to not back off. Retries are attempted immediately. Using a back off policy that causes threads to pause between attempts may cause performance issues, including excessive memory use and thread starvation. In high-volume environments, back off policies should be used with caution.

Configuring the Retry Advice

The examples in this section use the following <service-activator> that always throws an exception:

```
public class FailingService {

    public void service(String message) {
        throw new RuntimeException("error");
    }
}
```

Simple Stateless Retry

The default RetryTemplate has a SimpleRetryPolicy which tries three times. There is no BackOffPolicy, so the three attempts are made back-to-back-to-back with no delay between attempts. There is no RecoveryCallback, so the result is to throw the exception to the caller after the final failed retry occurs. In a Spring Integration environment, this final exception might be handled by using an error-channel on the inbound endpoint. The following example uses RetryTemplate and shows its DEBUG output:

```
<int:service-activator input-channel="input" ref="failer" method="service">
    <int:request-handler-advice-chain>
        <bean class="o.s.i.handler.advice.RequestHandlerRetryAdvice"/>
    </request-handler-advice-chain>
</int:service-activator>

DEBUG [task-scheduler-2]preSend on channel 'input', message: [Payload=...]
DEBUG [task-scheduler-2]Retry: count=0
DEBUG [task-scheduler-2]Checking for rethrow: count=1
DEBUG [task-scheduler-2]Retry: count=1
DEBUG [task-scheduler-2]Checking for rethrow: count=2
DEBUG [task-scheduler-2]Retry: count=2
DEBUG [task-scheduler-2]Checking for rethrow: count=3
DEBUG [task-scheduler-2]Retry failed last attempt: count=3
```

Simple Stateless Retry with Recovery

The following example adds a RecoveryCallback to the preceding example and uses an

`ErrorMessageSendingRecoverer` to send an `ErrorMessage` to a channel:

```
<int:service-activator input-channel="input" ref="failer" method="service">
    <int:request-handler-advice-chain>
        <bean class="o.s.i.handler.advice.RequestHandlerRetryAdvice">
            <property name="recoveryCallback">
                <bean class="o.s.i.handler.advice.ErrorMessageSendingRecoverer
">
                    <constructor-arg ref="myErrorChannel" />
                </bean>
            </property>
        </bean>
    </request-handler-advice-chain>
</int:int:service-activator>

DEBUG [task-scheduler-2]preSend on channel 'input', message: [Payload=...]
DEBUG [task-scheduler-2]Retry: count=0
DEBUG [task-scheduler-2]Checking for rethrow: count=1
DEBUG [task-scheduler-2]Retry: count=1
DEBUG [task-scheduler-2]Checking for rethrow: count=2
DEBUG [task-scheduler-2]Retry: count=2
DEBUG [task-scheduler-2]Checking for rethrow: count=3
DEBUG [task-scheduler-2]Retry failed last attempt: count=3
DEBUG [task-scheduler-2]Sending ErrorMessage :failedMessage:[Payload=...]
```

Stateless Retry with Customized Policies, and Recovery

For more sophistication, we can provide the advice with a customized `RetryTemplate`. This example continues to use the `SimpleRetryPolicy` but increases the attempts to four. It also adds an `ExponentialBackoffPolicy` where the first retry waits one second, the second waits five seconds and the third waits 25 (for four attempts in all). The following listing shows the example and its `DEBUG` output:

```
<int:service-activator input-channel="input" ref="failer" method="service">
    <int:request-handler-advice-chain>
        <bean class="o.s.i.handler.advice.RequestHandlerRetryAdvice">
            <property name="recoveryCallback">
                <bean class="o.s.i.handler.advice.ErrorMessageSendingRecoverer
">
                    <constructor-arg ref="myErrorChannel" />
                </bean>
            </property>
            <property name="retryTemplate" ref="retryTemplate" />
        </bean>
    </request-handler-advice-chain>
</int:service-activator>

<bean id="retryTemplate" class="
org.springframework.retry.support.RetryTemplate">
    <property name="retryPolicy">
        <bean class="org.springframework.retry.policy.SimpleRetryPolicy">
            <property name="maxAttempts" value="4" />
        </bean>
    </property>
    <property name="backOffPolicy">
        <bean class="
org.springframework.retry.backoff.ExponentialBackOffPolicy">
            <property name="initialInterval" value="1000" />
            <property name="multiplier" value="5.0" />
            <property name="maxInterval" value="60000" />
        </bean>
    </property>
</bean>

27.058 DEBUG [task-scheduler-1]preSend on channel 'input', message:
[Payload=...]
27.071 DEBUG [task-scheduler-1]Retry: count=0
27.080 DEBUG [task-scheduler-1]Sleeping for 1000
28.081 DEBUG [task-scheduler-1]Checking for rethrow: count=1
28.081 DEBUG [task-scheduler-1]Retry: count=1
28.081 DEBUG [task-scheduler-1]Sleeping for 5000
33.082 DEBUG [task-scheduler-1]Checking for rethrow: count=2
33.082 DEBUG [task-scheduler-1]Retry: count=2
33.083 DEBUG [task-scheduler-1]Sleeping for 25000
58.083 DEBUG [task-scheduler-1]Checking for rethrow: count=3
58.083 DEBUG [task-scheduler-1]Retry: count=3
58.084 DEBUG [task-scheduler-1]Checking for rethrow: count=4
58.084 DEBUG [task-scheduler-1]Retry failed last attempt: count=4
58.086 DEBUG [task-scheduler-1]Sending ErrorMessage
:failedMessage:[Payload=...]
```

Namespace Support for Stateless Retry

Starting with version 4.0, the preceding configuration can be greatly simplified, thanks to the namespace support for the retry advice, as the following example shows:

```
<int:service-activator input-channel="input" ref="failer" method="service">
    <int:request-handler-advice-chain>
        <bean ref="retrier" />
    </request-handler-advice-chain>
</int:service-activator>

<int:handler-retry-advice id="retrier" max-attempts="4" recovery-channel=
"myErrorChannel">
    <int:exponential-back-off initial="1000" multiplier="5.0" maximum="60000"
/>
</int:handler-retry-advice>
```

In the preceding example, the advice is defined as a top-level bean so that it can be used in multiple `request-handler-advice-chain` instances. You can also define the advice directly within the chain, as the following example shows:

```
<int:service-activator input-channel="input" ref="failer" method="service">
    <int:request-handler-advice-chain>
        <int:retry-advice id="retrier" max-attempts="4" recovery-channel=
"myErrorChannel">
            <int:exponential-back-off initial="1000" multiplier="5.0" maximum=
"60000" />
        </int:retry-advice>
    </request-handler-advice-chain>
</int:service-activator>
```

A `<handler-retry-advice>` can have a `<fixed-back-off>` or `<exponential-back-off>` child element or have no child element. A `<handler-retry-advice>` with no child element uses no back off. If there is no `recovery-channel`, the exception is thrown when retries are exhausted. The namespace can only be used with stateless retry.

For more complex environments (custom policies etc), use normal `<bean>` definitions.

Simple Stateful Retry with Recovery

To make retry stateful, we need to provide the advice with a `RetryStateGenerator` implementation. This class is used to identify a message as being a resubmission so that the `RetryTemplate` can determine the current state of retry for this message. The framework provides a `SpelExpressionRetryStateGenerator`, which determines the message identifier by using a SpEL expression. This example again uses the default policies (three attempts with no back off). As with stateless retry, these policies can be customized. The following listing shows the example

and its DEBUG output:

```
<int:service-activator input-channel="input" ref="failer" method="service">
    <int:request-handler-advice-chain>
        <bean class="o.s.i.handler.advice.RequestHandlerRetryAdvice">
            <property name="retryStateGenerator">
                <bean class=
"o.s.i.handler.advice.SpelExpressionRetryStateGenerator">
                    <constructor-arg value="headers['jms_messageId']" />
                </bean>
            </property>
            <property name="recoveryCallback">
                <bean class="o.s.i.handler.advice.ErrorMessageSendingRecoverer
">
                    <constructor-arg ref="myErrorChannel" />
                </bean>
            </property>
        </bean>
    </int:request-handler-advice-chain>
</int:service-activator>

24.351 DEBUG [Container#0-1]preSend on channel 'input', message: [Payload=...]
24.368 DEBUG [Container#0-1]Retry: count=0
24.387 DEBUG [Container#0-1]Checking for rethrow: count=1
24.387 DEBUG [Container#0-1]Rethrow in retry for policy: count=1
24.387 WARN  [Container#0-1]failure occurred in gateway sendAndReceive
org.springframework.integration.MessagingException: Failed to invoke handler
...
Caused by: java.lang.RuntimeException: foo
...
24.391 DEBUG [Container#0-1]Initiating transaction rollback on application
exception
...
25.412 DEBUG [Container#0-1]preSend on channel 'input', message: [Payload=...]
25.412 DEBUG [Container#0-1]Retry: count=1
25.413 DEBUG [Container#0-1]Checking for rethrow: count=2
25.413 DEBUG [Container#0-1]Rethrow in retry for policy: count=2
25.413 WARN  [Container#0-1]failure occurred in gateway sendAndReceive
org.springframework.integration.MessagingException: Failed to invoke handler
...
Caused by: java.lang.RuntimeException: foo
...
25.414 DEBUG [Container#0-1]Initiating transaction rollback on application
exception
...
26.418 DEBUG [Container#0-1]preSend on channel 'input', message: [Payload=...]
26.418 DEBUG [Container#0-1]Retry: count=2
26.419 DEBUG [Container#0-1]Checking for rethrow: count=3
26.419 DEBUG [Container#0-1]Rethrow in retry for policy: count=3
26.419 WARN  [Container#0-1]failure occurred in gateway sendAndReceive
org.springframework.integration.MessagingException: Failed to invoke handler
```

```
...
Caused by: java.lang.RuntimeException: foo
...
26.420 DEBUG [Container#0-1]Initiating transaction rollback on application
exception
...
27.425 DEBUG [Container#0-1]preSend on channel 'input', message: [Payload=...]
27.426 DEBUG [Container#0-1]Retry failed last attempt: count=3
27.426 DEBUG [Container#0-1]Sending ErrorMessage :failedMessage:[Payload=...]
```

If you compare the preceding example with the stateless examples, you can see that, with stateful retry, the exception is thrown to the caller on each failure.

Exception Classification for Retry

Spring Retry has a great deal of flexibility for determining which exceptions can invoke retry. The default configuration retries for all exceptions and the exception classifier looks at the top-level exception. If you configure it to, say, retry only on `MyException` and your application throws a `SomeOtherException` where the cause is a `MyException`, retry does not occur.

Since Spring Retry 1.0.3, the `BinaryExceptionClassifier` has a property called `traverseCauses` (the default is `false`). When `true`, it traverses exception causes until it finds a match or runs out of causes to traverse.

To use this classifier for retry, use a `SimpleRetryPolicy` created with the constructor that takes the max attempts, the `Map` of `Exception` objects, and the `traverseCauses` boolean. Then you can inject this policy into the `RetryTemplate`.

`traverseCauses` is required in this case because user exceptions may be wrapped in a `MessagingException`.

Circuit Breaker Advice

The general idea of the circuit breaker pattern is that, if a service is not currently available, do not waste time (and resources) trying to use it. The `o.s.i.handler.advice.RequestHandlerCircuitBreakerAdvice` implements this pattern. When the circuit breaker is in the closed state, the endpoint attempts to invoke the service. The circuit breaker goes to the open state if a certain number of consecutive attempts fail. When it is in the open state, new requests “fail fast” and no attempt is made to invoke the service until some time has expired.

When that time has expired, the circuit breaker is set to the half-open state. When in this state, if even a single attempt fails, the breaker immediately goes to the open state. If the attempt succeeds, the breaker goes to the closed state, in which case it does not go to the open state again until the configured number of consecutive failures again occur. Any successful attempt resets the state to zero failures for the purpose of determining when the breaker might go to the open state again.

Typically, this advice might be used for external services, where it might take some time to fail (such as a timeout attempting to make a network connection).

The `RequestHandlerCircuitBreakerAdvice` has two properties: `threshold` and `halfOpenAfter`. The `threshold` property represents the number of consecutive failures that need to occur before the breaker goes open. It defaults to `5`. The `halfOpenAfter` property represents the time after the last failure that the breaker waits before attempting another request. The default is 1000 milliseconds.

The following example configures a circuit breaker and shows its `DEBUG` and `ERROR` output:

```
<int:service-activator input-channel="input" ref="failer" method="service">
    <int:request-handler-advice-chain>
        <bean class="o.s.i.handler.advice.RequestHandlerCircuitBreakerAdvice">
            <property name="threshold" value="2" />
            <property name="halfOpenAfter" value="12000" />
        </bean>
    </int:request-handler-advice-chain>
</int:service-activator>

05.617 DEBUG [task-scheduler-1]preSend on channel 'input', message: [Payload=...]
05.638 ERROR [task-scheduler-
1]org.springframework.messaging.MessageHandlingException:
java.lang.RuntimeException: foo
...
10.598 DEBUG [task-scheduler-2]preSend on channel 'input', message: [Payload=...]
10.600 ERROR [task-scheduler-
2]org.springframework.messaging.MessageHandlingException:
java.lang.RuntimeException: foo
...
15.598 DEBUG [task-scheduler-3]preSend on channel 'input', message: [Payload=...]
15.599 ERROR [task-scheduler-3]org.springframework.messaging.MessagingException:
Circuit Breaker is Open for ServiceActivator
...
20.598 DEBUG [task-scheduler-2]preSend on channel 'input', message: [Payload=...]
20.598 ERROR [task-scheduler-2]org.springframework.messaging.MessagingException:
Circuit Breaker is Open for ServiceActivator
...
25.598 DEBUG [task-scheduler-5]preSend on channel 'input', message: [Payload=...]
25.601 ERROR [task-scheduler-
5]org.springframework.messaging.MessageHandlingException:
java.lang.RuntimeException: foo
...
30.598 DEBUG [task-scheduler-1]preSend on channel 'input', message:
[Payload=foo...]
30.599 ERROR [task-scheduler-1]org.springframework.messaging.MessagingException:
Circuit Breaker is Open for ServiceActivator
```

In the preceding example, the threshold is set to `2` and `halfOpenAfter` is set to `12` seconds. A new request arrives every 5 seconds. The first two attempts invoked the service. The third and fourth failed with an exception indicating that the circuit breaker is open. The fifth request was attempted because the request was 15 seconds after the last failure. The sixth attempt fails immediately

because the breaker immediately went to open.

Expression Evaluating Advice

The final supplied advice class is the `o.s.i.handler.advice.ExpressionEvaluatingRequestHandlerAdvice`. This advice is more general than the other two advices. It provides a mechanism to evaluate an expression on the original inbound message sent to the endpoint. Separate expressions are available to be evaluated, after either success or failure. Optionally, a message containing the evaluation result, together with the input message, can be sent to a message channel.

A typical use case for this advice might be with an `<ftp:outbound-channel-adapter/>`, perhaps to move the file to one directory if the transfer was successful or to another directory if it fails:

The advice has properties to set an expression when successful, an expression for failures, and corresponding channels for each. For the successful case, the message sent to the `successChannel` is an `AdviceMessage`, with the payload being the result of the expression evaluation. An additional property, called `inputMessage`, contains the original message sent to the handler. A message sent to the `failureChannel` (when the handler throws an exception) is an `ErrorMessage` with a payload of `MessageHandlingExpressionEvaluatingAdviceException`. Like all `MessagingException` instances, this payload has `failedMessage` and `cause` properties, as well as an additional property called `evaluationResult`, which contains the result of the expression evaluation.

Starting with version 5.1.3, if channels are configured, but expressions are not provided, the default expression is used to evaluate to the `payload` of the message.

When an exception is thrown in the scope of the advice, by default, that exception is thrown to the caller after any `failureExpression` is evaluated. If you wish to suppress throwing the exception, set the `trapException` property to `true`. The following advice shows how to configure an advice with Java DSL:

```
@SpringBootApplication
public class EerhaApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context = SpringApplication.run
(EerhaApplication.class, args);
        MessageChannel in = context.getBean("advised.input", MessageChannel.class
);
        in.send(new GenericMessage<>("good"));
        in.send(new GenericMessage<>("bad"));
        context.close();
    }

    @Bean
    public IntegrationFlow advised() {
        return f -> f.handle((GenericHandler<String>) (payload, headers) -> {
            if (payload.equals("good")) {
                return null;
            }
            else {
                throw new RuntimeException("some failure");
            }
        }, c -> c.advice(expressionAdvice()));
    }

    @Bean
    public Advice expressionAdvice() {
        ExpressionEvaluatingRequestHandlerAdvice advice = new
ExpressionEvaluatingRequestHandlerAdvice();
        advice.setSuccessChannelName("success.input");
        advice.setOnSuccessExpressionString("payload + ' was successful'");
        advice.setFailureChannelName("failure.input");
        advice.setOnFailureExpressionString(
                "payload + ' was bad, with reason: ' + #exception.cause.message");
        advice.setTrapException(true);
        return advice;
    }

    @Bean
    public IntegrationFlow success() {
        return f -> f.handle(System.out::println);
    }

    @Bean
    public IntegrationFlow failure() {
        return f -> f.handle(System.out::println);
    }

}
```

Rate Limiter Advice

The Rate Limiter advice (`RateLimiterRequestHandlerAdvice`) allows to ensure that an endpoint does not get overloaded with requests. When the rate limit is breached the request will go in a blocked state.

A typical use case for this advice might be an external service provider not allowing more than `n` number of request per minute.

The `RateLimiterRequestHandlerAdvice` implementation is fully based on the Resilience4j project and requires either `RateLimiter` or `RateLimiterConfig` injections. Can also be configured with defaults and/or custom name.

The following example configures a rate limiter advice with one request per 1 second:

```
@Bean
public RateLimiterRequestHandlerAdvice rateLimiterRequestHandlerAdvice() {
    return new RateLimiterRequestHandlerAdvice(RateLimiterConfig.custom()
            .limitRefreshPeriod(Duration.ofSeconds(1))
            .limitForPeriod(1)
            .build());
}

@ServiceActivator(inputChannel = "requestChannel", outputChannel = "resultChannel
",
        adviceChain = "rateLimiterRequestHandlerAdvice")
public String handleRequest(String payload) {
    ...
}
```

Caching Advice

Starting with version 5.2, the `CacheRequestHandlerAdvice` has been introduced. It is based on the caching abstraction in Spring Framework and aligned with the concepts and functionality provided by the `@Caching` annotation family. The logic internally is based on the `CacheAspectSupport` extension, where proxying for caching operations is done around the `AbstractReplyProducingMessageHandler.RequestHandler.handleRequestMessage` method with the request `Message<?>` as the argument. This advice can be configured with a SpEL expression or a `Function` to evaluate a cache key. The request `Message<?>` is available as the root object for the SpEL evaluation context, or as the `Function` input argument. By default, the `payload` of the request message is used for the cache key. The `CacheRequestHandlerAdvice` must be configured with `cacheNames`, when a default cache operation is a `CacheableOperation`, or with a set of any arbitrary `CacheOperation` s. Every `CacheOperation` can be configured separately or have shared options, like a `CacheManager`, `CacheResolver` and `CacheErrorHandler`, can be reused from the `CacheRequestHandlerAdvice` configuration. This configuration functionality is similar to Spring Framework's `@CacheConfig` and `@Caching` annotation combination. If a `CacheManager` is not provided, a single bean is resolved by default from the `BeanFactory` in the `CacheAspectSupport`.

The following example configures two advices with different set of caching operations:

```
@Bean
public CacheRequestHandlerAdvice cacheAdvice() {
    CacheRequestHandlerAdvice cacheRequestHandlerAdvice = new
CacheRequestHandlerAdvice(TEST_CACHE);
    cacheRequestHandlerAdvice.setKeyExpressionString("payload");
    return cacheRequestHandlerAdvice;
}

@Transformer(inputChannel = "transformerChannel", outputChannel = "nullChannel",
adviceChain = "cacheAdvice")
public Object transform(Message<?> message) {
    ...
}

@Bean
public CacheRequestHandlerAdvice cachePutAndEvictAdvice() {
    CacheRequestHandlerAdvice cacheRequestHandlerAdvice = new
CacheRequestHandlerAdvice();
    cacheRequestHandlerAdvice.setKeyExpressionString("payload");
    CachePutOperation.Builder cachePutBuilder = new CachePutOperation.Builder();
    cachePutBuilder.setCacheName(TEST_PUT_CACHE);
    CacheEvictOperation.Builder cacheEvictBuilder = new CacheEvictOperation
.Builder();
    cacheEvictBuilder.setCacheName(TEST_CACHE);
    cacheRequestHandlerAdvice.setCacheOperations(cachePutBuilder.build(),
cacheEvictBuilder.build());
    return cacheRequestHandlerAdvice;
}

@ServiceActivator(inputChannel = "serviceChannel", outputChannel = "nullChannel",
    adviceChain = "cachePutAndEvictAdvice")
public Message<?> service(Message<?> message) {
    ...
}
```

10.9.2. Reactive Advice

Starting with version 5.3, a `ReactiveRequestHandlerAdvice` can be used for request message handlers producing a `Mono` replies. A `BiFunction<Message<?>, Mono<?>, Publisher<?>>` has to be provided for this advice and it is called from the `Mono.transform()` operator on a reply produced by the intercepted `handleRequestMessage()` method implementation. Typically such a `Mono` customization is necessary when we would like to control network fluctuations via `timeout()`, `retry()` and similar support operators. For example when we can an HTTP request over WebFlux client, we could use below configuration to not wait for response more than 5 seconds:

```
.handle(WebFlux.outboundGateway("https://somehost/"),
                        e -> e.customizeMonoReply((message, mono) -> mono.timeout
(Duration.ofSeconds(5))));
```

The `message` argument is the request message for the message handler and can be used to determine request-scope attributes. The `mono` argument is the result of this message handler's `handleRequestMessage()` method implementation. A nested `Mono.transform()` can also be called from this function to apply, for example, a Reactive Circuit Breaker.

10.9.3. Custom Advice Classes

In addition to the provided advice classes described earlier, you can implement your own advice classes. While you can provide any implementation of `org.aopalliance.aop.Advice` (usually `org.aopalliance.intercept.MethodInterceptor`), we generally recommend that you subclass `o.s.i.handler.advice.AbstractRequestHandlerAdvice`. This has the benefit of avoiding the writing of low-level aspect-oriented programming code as well as providing a starting point that is specifically tailored for use in this environment.

Subclasses need to implement the `doInvoke()` method, the definition of which follows:

```
/**
 * Subclasses implement this method to apply behavior to the {@link
MessageHandler} callback.execute()
 * invokes the handler method and returns its result, or null).
 * @param callback Subclasses invoke the execute() method on this interface to
invoke the handler method.
 * @param target The target handler.
 * @param message The message that will be sent to the handler.
 * @return the result after invoking the {@link MessageHandler}.
 * @throws Exception
 */
protected abstract Object doInvoke(ExecutionCallback callback, Object target,
Message<?> message) throws Exception;
```

The callback parameter is a convenience to avoid subclasses that deal with AOP directly. Invoking the `callback.execute()` method invokes the message handler.

The `target` parameter is provided for those subclasses that need to maintain state for a specific handler, perhaps by maintaining that state in a `Map` keyed by the target. This feature allows the same advice to be applied to multiple handlers. The `RequestHandlerCircuitBreakerAdvice` uses advice this to keep circuit breaker state for each handler.

The `message` parameter is the message sent to the handler. While the advice cannot modify the message before invoking the handler, it can modify the payload (if it has mutable properties).

Typically, an advice would use the message for logging or to send a copy of the message somewhere before or after invoking the handler.

The return value would normally be the value returned by `callback.execute()`. However, the advice does have the ability to modify the return value. Note that only `AbstractReplyProducingMessageHandler` instances return values. The following example shows a custom advice class that extends `AbstractRequestHandlerAdvice`:

```
public class MyAdvice extends AbstractRequestHandlerAdvice {

    @Override
    protected Object doInvoke(ExecutionCallback callback, Object target, Message<
?> message) throws Exception {
        // add code before the invocation
        Object result = callback.execute();
        // add code after the invocation
        return result;
    }
}
```

In addition to the `execute()` method, `ExecutionCallback` provides an additional method: `cloneAndExecute()`. This method must be used in cases where the invocation might be called multiple times within a single execution of `doInvoke()`, such as in the `RequestHandlerRetryAdvice`. This is required because the Spring AOP `org.springframework.aop.framework.ReflectiveMethodInvocation` object maintains state by keeping track of which advice in a chain was last invoked. This state must be reset for each call.

For more information, see the ReflectiveMethodInvocation Javadoc.

10.9.4. Other Advice Chain Elements

While the abstract class mentioned above is a convenience, you can add any `Advice`, including a transaction advice, to the chain.

10.9.5. Handling Message Advice

As discussed in the introduction to this section, advice objects in a request handler advice chain are applied to just the current endpoint, not the downstream flow (if any). For `MessageHandler` objects that produce a reply (such as those that extend `AbstractReplyProducingMessageHandler`), the advice is applied to an internal method: `handleRequestMessage()` (called from `MessageHandler.handleMessage()`). For other message handlers, the advice is applied to `MessageHandler.handleMessage()`.

There are some circumstances where, even if a message handler is an `AbstractReplyProducingMessageHandler`, the advice must be applied to the `handleMessage` method. For

example, the idempotent receiver might return `null`, which would cause an exception if the handler's `replyRequired` property is set to `true`. Another example is the `BoundRabbitChannelAdvice` — see Strict Message Ordering.

Starting with version 4.3.1, a new `HandleMessageAdvice` interface and its base implementation (`AbstractHandleMessageAdvice`) have been introduced. `Advice` objects that implement `HandleMessageAdvice` are always applied to the `handleMessage()` method, regardless of the handler type.

It is important to understand that `HandleMessageAdvice` implementations (such as idempotent receiver), when applied to a handlers that return responses, are dissociated from the `adviceChain` and properly applied to the `MessageHandler.handleMessage()` method.

Because of this disassociation, the advice chain order is not honored.

Consider the following configuration:

```
<some-reply-producing-endpoint ... >
    <int:request-handler-advice-chain>
        <tx:advice ... />
        <bean ref="myHandleMessageAdvice" />
    </int:request-handler-advice-chain>
</some-reply-producing-endpoint>
```

In the preceding example, the `<tx:advice>` is applied to the `AbstractReplyProducingMessageHandler.handleRequestMessage()`. However, `myHandleMessageAdvice` is applied for to `MessageHandler.handleMessage()`. Therefore, it is invoked **before** the `<tx:advice>`. To retain the order, you should follow the standard Spring AOP configuration approach and use an endpoint `id` together with the `.handler` suffix to obtain the target `MessageHandler` bean. Note that, in that case, the entire downstream flow is within the transaction scope.

In the case of a `MessageHandler` that does not return a response, the advice chain order is retained.

Starting with version 5.3, the `HandleMessageAdviceAdapter` is present to let apply any existing `MethodInterceptor` for the `MessageHandler.handleMessage()` and, therefore, whole sub-flow. For example a `RetryOperationsInterceptor` could be applied for the whole sub-flow starting from some endpoint, which is not possible by default because consumer endpoint applies advices only for the `AbstractReplyProducingMessageHandler.RequestHandler.handleRequestMessage()`. Starting with version 5.3, the `HandleMessageAdviceAdapter` is provided to apply any `MethodInterceptor` for the `MessageHandler.handleMessage()` method and, therefore, the whole sub-flow. For example, a `RetryOperationsInterceptor` could be applied to the whole sub-flow starting from some endpoint; this is not possible, by default, because the consumer endpoint applies advices only to the `AbstractReplyProducingMessageHandler.RequestHandler.handleRequestMessage()`.

10.9.6. Transaction Support

Starting with version 5.0, a new `TransactionHandleMessageAdvice` has been introduced to make the whole downstream flow transactional, thanks to the `HandleMessageAdvice` implementation. When a regular `TransactionInterceptor` is used in the `<request-handler-advice-chain>` element (for example, through configuring `<tx:advice>`), a started transaction is only applied only for an internal `AbstractReplyProducingMessageHandler.handleRequestMessage()` and is not propagated to the downstream flow.

To simplify XML configuration, along with the `<request-handler-advice-chain>`, a `<transactional>` element has been added to all `<outbound-gateway>` and `<service-activator>` and related components. The following example shows `<transactional>` in use:

```
<int-rmi:outbound-gateway remote-channel="foo" host="localhost"
    request-channel="good" reply-channel="reply" port="#{@port}">
        <int-rmi:transactional/>
</int-rmi:outbound-gateway>

<bean id="transactionManager" class="org.mockito.Mockito" factory-method="mock">
    <constructor-arg value="org.springframework.transaction.TransactionManager"/>
</bean>
```

If you are familiar with the JPA integration components, such a configuration is not new, but now we can start a transaction from any point in our flow — not only from the `<poller>` or a message-driven channel adapter such as JMS.

Java configuration can be simplified by using the `TransactionInterceptorBuilder`, and the result bean name can be used in the messaging annotations `adviceChain` attribute, as the following example shows:

```
@Bean
public ConcurrentMetadataStore store() {
    return new SimpleMetadataStore(hazelcastInstance()
                        .getMap("idempotentReceiverMetadataStore"));
}

@Bean
public IdempotentReceiverInterceptor idempotentReceiverInterceptor() {
    return new IdempotentReceiverInterceptor(
            new MetadataStoreSelector(
                    message -> message.getPayload().toString(),
                    message -> message.getPayload().toString().toUpperCase(), store()
));
}

@Bean
public TransactionInterceptor transactionInterceptor() {
    return new TransactionInterceptorBuilder(true)
                .transactionManager(this.transactionManager)
                .isolation(Isolation.READ_COMMITTED)
                .propagation(Propagation.REQUIRES_NEW)
                .build();
}

@Bean
@org.springframework.integration.annotation.Transformer(inputChannel = "input",
          outputChannel = "output",
          adviceChain = { "idempotentReceiverInterceptor",
                  "transactionInterceptor" })
public Transformer transformer() {
    return message -> message;
}
```

Note the `true` parameter on the `TransactionInterceptorBuilder` constructor. It causes the creation of a `TransactionHandleMessageAdvice`, not a regular `TransactionInterceptor`.

Java DSL supports an `Advice` through the `.transactional()` options on the endpoint configuration, as the following example shows:

```
@Bean
public IntegrationFlow updatingGatewayFlow() {
    return f -> f
        .handle(Jpa.updatingGateway(this.entityManagerFactory),
                e -> e.transactional(true))
        .channel(c -> c.queue("persistResults"));
}
```

10.9.7. Advising Filters

There is an additional consideration when advising `Filter` advices. By default, any discard actions (when the filter returns `false`) are performed within the scope of the advice chain. This could include all the flow downstream of the discard channel. So, for example, if an element downstream of the discard channel throws an exception and there is a retry advice, the process is retried. Also, if `throwExceptionOnRejection` is set to `true` (the exception is thrown within the scope of the advice).

Setting `discard-within-advice` to `false` modifies this behavior and the discard (or exception) occurs after the advice chain is called.

10.9.8. Advising Endpoints Using Annotations

When configuring certain endpoints by using annotations (`@Filter`, `@ServiceActivator`, `@Splitter`, and `@Transformer`), you can supply a bean name for the advice chain in the `adviceChain` attribute. In addition, the `@Filter` annotation also has the `discardWithinAdvice` attribute, which can be used to configure the discard behavior, as discussed in Advising Filters. The following example causes the discard to be performed after the advice:

```
@MessageEndpoint
public class MyAdvisedFilter {

    @Filter(inputChannel="input", outputChannel="output",
            adviceChain="adviceChain", discardWithinAdvice="false")
    public boolean filter(String s) {
        return s.contains("good");
    }
}
```

10.9.9. Ordering Advices within an Advice Chain

Advice classes are “around” advices and are applied in a nested fashion. The first advice is the outermost, while the last advice is the innermost (that is, closest to the handler being advised). It is important to put the advice classes in the correct order to achieve the functionality you desire.

For example, suppose you want to add a retry advice and a transaction advice. You may want to place the retry advice advice first, followed by the transaction advice. Consequently, each retry is performed in a new transaction. On the other hand, if you want all the attempts and any recovery operations (in the retry `RecoveryCallback`) to be scoped within the transaction, you could put the transaction advice first.

10.9.10. Advised Handler Properties

Sometimes, it is useful to access handler properties from within the advice. For example, most handlers implement `NamedComponent` to let you access the component name.

The target object can be accessed through the `target` argument (when subclassing `AbstractRequestHandlerAdvice`) or `invocation.getThis()` (when implementing

`org.aopalliance.intercept.MethodInterceptor`).

When the entire handler is advised (such as when the handler does not produce replies or the advice implements `HandleMessageAdvice`), you can cast the target object to an interface, such as `NamedComponent`, as shown in the following example:

```
String componentName = ((NamedComponent) target).getComponentName();
```

When you implement `MethodInterceptor` directly, you could cast the target object as follows:

```
String componentName = ((NamedComponent) invocation.getThis()).getComponentName();
```

When only the `handleRequestMessage()` method is advised (in a reply-producing handler), you need to access the full handler, which is an `AbstractReplyProducingMessageHandler`. The following example shows how to do so:

```
AbstractReplyProducingMessageHandler handler =
    ((AbstractReplyProducingMessageHandler.RequestHandler) target)
.getAdvisedHandler();

String componentName = handler.getComponentName();
```

10.9.11. Idempotent Receiver Enterprise Integration Pattern

Starting with version 4.1, Spring Integration provides an implementation of the Idempotent Receiver Enterprise Integration Pattern. It is a functional pattern and the whole idempotency logic should be implemented in the application. However, to simplify the decision-making, the `IdempotentReceiverInterceptor` component is provided. This is an AOP `Advice` that is applied to the `MessageHandler.handleMessage()` method and that can `filter` a request message or mark it as a `duplicate`, according to its configuration.

Previously, you could have implemented this pattern by using a custom `MessageSelector` in a `<filter/>` (see Filter), for example. However, since this pattern really defines the behavior of an endpoint rather than being an endpoint itself, the idempotent receiver implementation does not provide an endpoint component. Rather, it is applied to endpoints declared in the application.

The logic of the `IdempotentReceiverInterceptor` is based on the provided `MessageSelector` and, if the message is not accepted by that selector, it is enriched with the `duplicateMessage` header set to `true`. The target `MessageHandler` (or downstream flow) can consult this header to implement the correct idempotency logic. If the `IdempotentReceiverInterceptor` is configured with a `discardChannel` or `throwExceptionOnRejection = true`, the duplicate message is not sent to the target

`MessageHandler.handleMessage()`. Rather, it is discarded. If you want to discard (do nothing with) the duplicate message, the `discardChannel` should be configured with a `NullChannel`, such as the default `nullChannel` bean.

To maintain state between messages and provide the ability to compare messages for the idempotency, we provide the `MetadataStoreSelector`. It accepts a `MessageProcessor` implementation (which creates a lookup key based on the `Message`) and an optional `ConcurrentMetadataStore` (Metadata Store). See the `MetadataStoreSelector` Javadoc for more information. You can also customize the `value` for `ConcurrentMetadataStore` by using an additional `MessageProcessor`. By default, `MetadataStoreSelector` uses the `timestamp` message header.

Normally, the selector selects a message for acceptance if there is no existing value for the key. In some cases, it is useful to compare the current and new values for a key, to determine whether the message should be accepted. Starting with version 5.3, the `compareValues` property is provided which references a `BiPredicate<String, String>`; the first parameter is the old value; return `true` to accept the message and replace the old value with the new value in the `MetadataStore`. This can be useful to reduce the number of keys; for example, when processing lines in a file, you can store the file name in the key and the current line number in the value. Then, after a restart, you can skip lines that have already been processed. See Idempotent Downstream Processing a Split File for an example.

For convenience, the `MetadataStoreSelector` options are configurable directly on the `<idempotent-receiver>` component. The following listing shows all the possible attributes:

```
<idempotent-receiver
        id=""  ①
        endpoint=""  ②
        selector=""  ③
        discard-channel=""  ④
        metadata-store=""  ⑤
        key-strategy=""  ⑥
        key-expression=""  ⑦
        value-strategy=""  ⑧
        value-expression=""  ⑨
        compare-values="" ⑩
        throw-exception-on-rejection="" />  ⑪
```

① The ID of the `IdempotentReceiverInterceptor` bean. Optional.

② Consumer endpoint name(s) or pattern(s) to which this interceptor is applied. Separate names (patterns) with commas (`,`), such as `endpoint="aaa, bbb*, ccc, *ddd, eee*fff"`. Endpoint bean names matching these patterns are then used to retrieve the target endpoint's `MessageHandler` bean (using its `.handler` suffix), and the `IdempotentReceiverInterceptor` is applied to those beans. Required.

③ A `MessageSelector` bean reference. Mutually exclusive with `metadata-store` and `key-strategy (key-expression)`. When `selector` is not provided, one of `key-strategy` or `key-strategy-expression` is required.

④ Identifies the channel to which to send a message when the `IdempotentReceiverInterceptor` does not accept it. When omitted, duplicate messages are forwarded to the handler with a `duplicateMessage` header. Optional.

⑤ A `ConcurrentMetadataStore` reference. Used by the underlying `MetadataStoreSelector`. Mutually exclusive with `selector`. Optional. The default `MetadataStoreSelector` uses an internal `SimpleMetadataStore` that does not maintain state across application executions.

⑥ A `MessageProcessor` reference. Used by the underlying `MetadataStoreSelector`. Evaluates an `idempotentKey` from the request message. Mutually exclusive with `selector` and `key-expression`. When a `selector` is not provided, one of `key-strategy` or `key-strategy-expression` is required.

⑦ A SpEL expression to populate an `ExpressionEvaluatingMessageProcessor`. Used by the underlying `MetadataStoreSelector`. Evaluates an `idempotentKey` by using the request message as the evaluation context root object. Mutually exclusive with `selector` and `key-strategy`. When a `selector` is not provided, one of `key-strategy` or `key-strategy-expression` is required.

⑧ A `MessageProcessor` reference. Used by the underlying `MetadataStoreSelector`. Evaluates a `value` for the `idempotentKey` from the request message. Mutually exclusive with `selector` and `value-expression`. By default, the 'MetadataStoreSelector' uses the 'timestamp' message header as the Metadata 'value'.

⑨ A SpEL expression to populate an `ExpressionEvaluatingMessageProcessor`. Used by the underlying `MetadataStoreSelector`. Evaluates a `value` for the `idempotentKey` by using the request message as the evaluation context root object. Mutually exclusive with `selector` and

`value-strategy`. By default, the 'MetadataStoreSelector' uses the 'timestamp' message header as the metadata 'value'.

⑩ A reference to a `BiPredicate<String, String>` bean which allows you to optionally select a message by comparing the old and new values for the key; `null` by default.

⑪ Whether to throw an exception if the `IdempotentReceiverInterceptor` rejects the message. Defaults to `false`. It is applied regardless of whether or not a `discard-channel` is provided.

For Java configuration, Spring Integration provides the method-level `@IdempotentReceiver` annotation. It is used to mark a `method` that has a messaging annotation (`@ServiceActivator`, `@Router`, `and others) to specify which `IdempotentReceiverInterceptor` objects are applied to this endpoint. The following example shows how to use the `@IdempotentReceiver` annotation:

```
@Bean
public IdempotentReceiverInterceptor idempotentReceiverInterceptor() {
   return new IdempotentReceiverInterceptor(new MetadataStoreSelector(m ->
                                                    m.getHeaders().get
(INVOICE_NBR_HEADER)));
}

@Bean
@ServiceActivator(inputChannel = "input", outputChannel = "output")
@IdempotentReceiver("idempotentReceiverInterceptor")
public MessageHandler myService() {
    ....
}
```

When you use the Java DSL, you can add the interceptor to the endpoint's advice chain, as the following example shows:

```
@Bean
public IntegrationFlow flow() {
    ...
        .handle("someBean", "someMethod",
            e -> e.advice(idempotentReceiverInterceptor()))
    ...
}
```

The `IdempotentReceiverInterceptor` is designed only for the `MessageHandler.handleMessage(Message<?>)` method. Starting with version 4.3.1, it implements `HandleMessageAdvice`, with the `AbstractHandleMessageAdvice` as a base class, for better dissociation. See Handling Message Advice for more information.

10.10. Logging Channel Adapter

The `<logging-channel-adapter>` is often used in conjunction with a wire tap, as discussed in Wire Tap. However, it can also be used as the ultimate consumer of any flow. For example, consider a flow that ends with a `<service-activator>` that returns a result, but you wish to discard that result. To do that, you could send the result to `NullChannel`. Alternatively, you can route it to an `INFO` level `<logging-channel-adapter>`. That way, you can see the discarded message when logging at `INFO` level but not see it when logging at (for example) the `WARN` level. With a `NullChannel`, you would see only the discarded message when logging at the `DEBUG` level. The following listing shows all the possible attributes for the `logging-channel-adapter` element:

```
<int:logging-channel-adapter
    channel="" ①
    level="INFO" ②
    expression="" ③
    log-full-message="false" ④
    logger-name="" /> ⑤
```

① The channel connecting the logging adapter to an upstream component.

② The logging level at which messages sent to this adapter will be logged. Default: `INFO`.

③ A SpEL expression representing exactly what parts of the message are logged. Default: `payload` — only the payload is logged. if `log-full-message` is specified, this attribute cannot be specified.

④ When `true`, the entire message (including headers) is logged. Default: `false` — only the payload is logged. This attribute cannot be specified if `expression` is specified.

⑤ Specifies the `name` of the logger (known as `category` in `log4j`). Used to identify log messages created by this adapter. This enables setting the log name (in the logging subsystem) for individual adapters. By default, all adapters log under the following name: `org.springframework.integration.handler.LoggingHandler`.

10.10.1. Using Java Configuration

The following Spring Boot application shows an example of configuring the `LoggingHandler` by using Java configuration:

```
@SpringBootApplication
public class LoggingJavaApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
             new SpringApplicationBuilder(LoggingJavaApplication.class)
                    .web(false)
                    .run(args);
         MyGateway gateway = context.getBean(MyGateway.class);
         gateway.sendToLogger("foo");
    }

    @Bean
    @ServiceActivator(inputChannel = "logChannel")
    public LoggingHandler logging() {
        LoggingHandler adapter = new LoggingHandler(LoggingHandler.Level.DEBUG);
        adapter.setLoggerName("TEST_LOGGER");
        adapter.setLogExpressionString("headers.id + ': ' + payload");
        return adapter;
    }

    @MessagingGateway(defaultRequestChannel = "logChannel")
    public interface MyGateway {

        void sendToLogger(String data);

    }

}
```

10.10.2. Configuring with the Java DSL

The following Spring Boot application shows an example of configuring the logging channel adapter by using the Java DSL:

```
@SpringBootApplication
public class LoggingJavaApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
             new SpringApplicationBuilder(LoggingJavaApplication.class)
                    .web(false)
                    .run(args);
         MyGateway gateway = context.getBean(MyGateway.class);
         gateway.sendToLogger("foo");
    }

    @Bean
    public IntegrationFlow loggingFlow() {
        return IntegrationFlows.from(MyGateway.class)
                     .log(LoggingHandler.Level.DEBUG, "TEST_LOGGER",
                           m -> m.getHeaders().getId() + ": " + m.getPayload());
    }

    @MessagingGateway
    public interface MyGateway {

        void sendToLogger(String data);

    }

}
```

10.11. `java.util.function` Interfaces Support

Starting with version 5.1, Spring Integration provides direct support for interfaces in the `java.util.function` package. All messaging endpoints, (Service Activator, Transformer, Filter, etc.) can now refer to `Function` (or `Consumer`) beans. The Messaging Annotations can be applied directly on these beans similar to regular `MessageHandler` definitions. For example if you have this `Function` bean definition:

```
@Configuration
public class FunctionConfiguration {

    @Bean
    public Function<String, String> functionAsService() {
        return String::toUpperCase;
    }

}
```

You can use it as a simple reference in an XML configuration file:

```
<service-activator input-channel="processorViaFunctionChannel" ref=
"functionAsService"/>
```

When we configure our flow with Messaging Annotations, the code is straightforward:

```
@Bean
@Transformer(inputChannel = "functionServiceChannel")
public Function<String, String> functionAsService() {
    return String::toUpperCase;
}
```

When the function returns an array, `Collection` (essentially, any `Iterable`), `Stream` or Reactor `Flux`, `@Splitter` can be used on such a bean to perform iteration over the result content.

The `java.util.function.Consumer` interface can be used for an `<int:outbound-channel-adapter>` or, together with the `@ServiceActivator` annotation, to perform the final step of a flow:

```
@Bean
@ServiceActivator(inputChannel = "messageConsumerServiceChannel")
public Consumer<Message<?>> messageConsumerAsService() {
    // Has to be an anonymous class for proper type inference
    return new Consumer<Message<?>>() {

        @Override
        public void accept(Message<?> e) {
            collector().add(e);
        }

    };
}
```

Also, pay attention to the comment in the code snippet above: if you would like to deal with the whole message in your `Function`/`Consumer` you cannot use a lambda definition. Because of Java type erasure we cannot determine the target type for the `apply()`/`accept()` method call.

The `java.util.function.Supplier` interface can simply be used together with the `@InboundChannelAdapter` annotation, or as a `ref` in an `<int:inbound-channel-adapter>`:

```
@Bean
@InboundChannelAdapter(value = "inputChannel", poller = @Poller(fixedDelay = "
1000"))
public Supplier<String> pojoSupplier() {
    return () -> "foo";
}
```

With the Java DSL we just need to use a reference to the function bean in the endpoint definitions. Meanwhile an implementation of the `Supplier` interface can be used as regular `MessageSource` definition:

```
@Bean
public Function<String, String> toUpperCaseFunction() {
    return String::toUpperCase;
}

@Bean
public Supplier<String> stringSupplier() {
    return () -> "foo";
}

@Bean
public IntegrationFlow supplierFlow() {
    return IntegrationFlows.from(stringSupplier())
                .transform(toUpperCaseFunction())
                .channel("suppliedChannel")
                .get();
}
```

This function support is useful when used together with the Spring Cloud Function framework, where we have a function catalog and can refer to its member functions from an integration flow definition.

10.11.1. Kotlin Lambdas

The Framework also has been improved to support Kotlin lambdas for functions so now you can use a combination of the Kotlin language and Spring Integration flow definitions:

```
@Bean
@Transformer(inputChannel = "functionServiceChannel")
fun kotlinFunction(): (String) -> String {
    return { it.toUpperCase() }
}

@Bean
@ServiceActivator(inputChannel = "messageConsumerServiceChannel")
fun kotlinConsumer(): (Message<Any>) -> Unit {
    return { print(it) }
}

@Bean
@InboundChannelAdapter(value = "counterChannel",
        poller = [Poller(fixedRate = "10", maxMessagesPerPoll = "1")])
fun kotlinSupplier(): () -> String {
    return { "baz" }
}
```

Chapter 11. Java DSL

The Spring Integration Java configuration and DSL provides a set of convenient builders and a fluent API that lets you configure Spring Integration message flows from Spring `@Configuration` classes.

(See also Kotlin DSL.)

The Java DSL for Spring Integration is essentially a facade for Spring Integration. The DSL provides a simple way to embed Spring Integration Message Flows into your application by using the fluent `Builder` pattern together with existing Java configuration from Spring Framework and Spring Integration. We also use and support lambdas (available with Java 8) to further simplify Java configuration.

The cafe offers a good example of using the DSL.

The DSL is presented by the `IntegrationFlows` factory for the `IntegrationFlowBuilder`. This produces the `IntegrationFlow` component, which should be registered as a Spring bean (by using the `@Bean` annotation). The builder pattern is used to express arbitrarily complex structures as a hierarchy of methods that can accept lambdas as arguments.

The `IntegrationFlowBuilder` only collects integration components (`MessageChannel` instances, `AbstractEndpoint` instances, and so on) in the `IntegrationFlow` bean for further parsing and registration of concrete beans in the application context by the `IntegrationFlowBeanPostProcessor`.

The Java DSL uses Spring Integration classes directly and bypasses any XML generation and parsing. However, the DSL offers more than syntactic sugar on top of XML. One of its most compelling features is the ability to define inline lambdas to implement endpoint logic, eliminating the need for external classes to implement custom logic. In some sense, Spring Integration's support for the Spring Expression Language (SpEL) and inline scripting address this, but lambdas are easier and much more powerful.

The following example shows how to use Java Configuration for Spring Integration:

```
@Configuration
@EnableIntegration
public class MyConfiguration {

    @Bean
    public AtomicInteger integerSource() {
        return new AtomicInteger();
    }

    @Bean
    public IntegrationFlow myFlow() {
        return IntegrationFlows.from(integerSource::getAndIncrement,
                                     c -> c.poller(Pollers.fixedRate(100)))
                .channel("inputChannel")
                .filter((Integer p) -> p > 0)
                .transform(Object::toString)
                .channel(MessageChannels.queue())
                .get();
    }
}
```

The result of the preceding configuration example is that it creates, after `ApplicationContext` start up, Spring Integration endpoints and message channels. Java configuration can be used both to replace and augment XML configuration. You need not replace all of your existing XML configuration to use Java configuration.

11.1. DSL Basics

The `org.springframework.integration.dsl` package contains the `IntegrationFlowBuilder` API mentioned earlier and a number of `IntegrationComponentSpec` implementations, which are also builders and provide the fluent API to configure concrete endpoints. The `IntegrationFlowBuilder` infrastructure provides common enterprise integration patterns (EIP) for message-based applications, such as channels, endpoints, pollers, and channel interceptors.

Endpoints are expressed as verbs in the DSL to improve readability. The following list includes the common DSL method names and the associated EIP endpoint:

- transform → `Transformer`
- filter → `Filter`
- handle → `ServiceActivator`
- split → `Splitter`
- aggregate → `Aggregator`
- route → `Router`
- bridge → `Bridge`

Conceptually, integration processes are constructed by composing these endpoints into one or more message flows. Note that EIP does not formally define the term 'message flow', but it is useful to think of it as a unit of work that uses well known messaging patterns. The DSL provides an `IntegrationFlow` component to define a composition of channels and endpoints between them, but now `IntegrationFlow` plays only the configuration role to populate real beans in the application context and is not used at runtime. The following example uses the `IntegrationFlows` factory to define an `IntegrationFlow` bean by using EIP-methods from `IntegrationFlowBuilder`:

```
@Bean
public IntegrationFlow integerFlow() {
    return IntegrationFlows.from("input")
            .<String, Integer>transform(Integer::parseInt)
            .get();
}
```

The `transform` method accepts a lambda as an endpoint argument to operate on the message payload. The real argument of this method is `GenericTransformer<S, T>`. Consequently, any of the provided transformers (`ObjectToJsonTransformer`, `FileToStringTransformer`, and other) can be used here.

Under the covers, `IntegrationFlowBuilder` recognizes the `MessageHandler` and the endpoint for it, with `MessageTransformingHandler` and `ConsumerEndpointFactoryBean`, respectively. Consider another example:

```
@Bean
public IntegrationFlow myFlow() {
    return IntegrationFlows.from("input")
                .filter("World"::equals)
                .transform("Hello "::concat)
                .handle(System.out::println)
                .get();
}
```

The preceding example composes a sequence of `Filter -> Transformer -> Service Activator`. The flow is "'one way'". That is, it does not provide a reply message but only prints the payload to STDOUT. The endpoints are automatically wired together by using direct channels.

Lambdas And `Message<?>` Arguments

When using lambdas in EIP methods, the "input" argument is generally the message payload. If you wish to access the entire message, use one of the overloaded methods that take a `Class<?>` as the first parameter. For example, this won't work:

```
.<Message<?>, Foo>transform(m -> newFooFromMessage(m))
```

This will fail at runtime with a `ClassCastException` because the lambda doesn't retain the argument type and the framework will attempt to cast the payload to a `Message<?>`.

Instead, use:

```
.(Message.class, m -> newFooFromMessage(m))
```

Bean Definitions override

The Java DSL can register beans for the object defined in-line in the flow definition, as well as can reuse existing, injected beans. In case of the same bean name defined for in-line object and existing bean definition, a `BeanDefinitionOverrideException` is thrown indicating that such a configuration is wrong. However when you deal with `prototype` beans, there is no way to detect from the integration flow processor an existing bean definition because every time we call a `prototype` bean from the `BeanFactory` we get a new instance. This way a provided instance is used in the `IntegrationFlow` as is without any bean registration and any possible check against existing `prototype` bean definition. However `BeanFactory.initializeBean()` is called for this object if it has an explicit `id` and bean definition for this name is in `prototype` scope.

11.2. Message Channels

In addition to the `IntegrationFlowBuilder` with EIP methods, the Java DSL provides a fluent API to configure `MessageChannel` instances. For this purpose the `MessageChannels` builder factory is provided. The following example shows how to use it:

```
@Bean
public MessageChannel priorityChannel() {
    return MessageChannels.priority(this.mongoDbChannelMessageStore,
"priorityGroup")
                          .interceptor(wireTap())
                          .get();
}
```

The same `MessageChannels` builder factory can be used in the `channel()` EIP method from `IntegrationFlowBuilder` to wire endpoints, similar to wiring an `input-channel`/`output-channel` pair in the XML configuration. By default, endpoints are wired with `DirectChannel` instances where the bean name is based on the following pattern: `[IntegrationFlow.beanName].channel#[channelNameIndex]`. This rule is also applied for unnamed channels produced by inline `MessageChannels` builder factory usage. However all `MessageChannels` methods have a variant that is aware of the `channelId` that you can use to set the bean names for `MessageChannel` instances. The `MessageChannel` references and `beanName` can be used as bean-method invocations. The following example shows the possible ways to use the `channel()` EIP method:

```
@Bean
public MessageChannel queueChannel() {
    return MessageChannels.queue().get();
}

@Bean
public MessageChannel publishSubscribe() {
    return MessageChannels.publishSubscribe().get();
}

@Bean
public IntegrationFlow channelFlow() {
    return IntegrationFlows.from("input")
                .fixedSubscriberChannel()
                .channel("queueChannel")
                .channel(publishSubscribe())
                .channel(MessageChannels.executor("executorChannel", this.
taskExecutor))
                .channel("output")
                .get();
}
```

- `from("input")` means "'find and use the `MessageChannel` with the "input" id, or create one'".
- `fixedSubscriberChannel()` produces an instance of `FixedSubscriberChannel` and registers it with a name of `channelFlow.channel#0`.
- `channel("queueChannel")` works the same way but uses an existing `queueChannel` bean.
- `channel(publishSubscribe())` is the bean-method reference.
- `channel(MessageChannels.executor("executorChannel", this.taskExecutor))` is the `IntegrationFlowBuilder` that exposes `IntegrationComponentSpec` to the `ExecutorChannel` and registers it as `executorChannel`.
- `channel("output")` registers the `DirectChannel` bean with `output` as its name, as long as no beans with this name already exist.

Note: The preceding `IntegrationFlow` definition is valid, and all of its channels are applied to endpoints with `BridgeHandler` instances.

Be careful to use the same inline channel definition through `MessageChannels` factory from different `IntegrationFlow` instances. Even if the DSL parser registers non-existent objects as beans, it cannot determine the same object (`MessageChannel`) from different `IntegrationFlow` containers. The following example is wrong:

```
@Bean
public IntegrationFlow startFlow() {
    return IntegrationFlows.from("input")
                .transform(...)
                .channel(MessageChannels.queue("queueChannel"))
                .get();
}

@Bean
public IntegrationFlow endFlow() {
    return IntegrationFlows.from(MessageChannels.queue("queueChannel"))
                .handle(...)
                .get();
}
```

The result of that bad example is the following exception:

```
Caused by: java.lang.IllegalStateException:
Could not register object [queueChannel] under bean name 'queueChannel':
     there is already object [queueChannel] bound
        at
o.s.b.f.s.DefaultSingletonBeanRegistry.registerSingleton(DefaultSingletonBeanRegistry.
java:129)
```

To make it work, you need to declare `@Bean` for that channel and use its bean method from different `IntegrationFlow` instances.

11.3. Pollers

Spring Integration also provides a fluent API that lets you configure `PollerMetadata` for `AbstractPollingEndpoint` implementations. You can use the `Pollers` builder factory to configure common bean definitions or those created from `IntegrationFlowBuilder` EIP methods, as the following example shows:

```
@Bean(name = PollerMetadata.DEFAULT_POLLER)
public PollerSpec poller() {
    return Pollers.fixedRate(500)
        .errorChannel("myErrors");
}
```

See `Pollers` and `PollerSpec` in the Javadoc for more information.

If you use the DSL to construct a `PollerSpec` as a `@Bean`, do not call the `get()` method in the bean definition. The `PollerSpec` is a `FactoryBean` that generates the `PollerMetadata` object from the specification and initializes all of its properties.

11.4. DSL and Endpoint Configuration

All `IntegrationFlowBuilder` EIP methods have a variant that applies the lambda parameter to provide options for `AbstractEndpoint` instances: `SmartLifecycle`, `PollerMetadata`, `request-handler-advice-chain`, and others. Each of them has generic arguments, so it lets you configure an endpoint and even its `MessageHandler` in the context, as the following example shows:

```
@Bean
public IntegrationFlow flow2() {
    return IntegrationFlows.from(this.inputChannel)
                .transform(new PayloadSerializingTransformer(),
                       c -> c.autoStartup(false).id("
payloadSerializingTransformer"))
                .transform((Integer p) -> p * 2, c -> c.advice(this
.expressionAdvice()))
                .get();
}
```

In addition, the `EndpointSpec` provides an `id()` method to let you register an endpoint bean with a given bean name, rather than a generated one.

If the `MessageHandler` is referenced as a bean, then any existing `adviceChain` configuration will be overridden if the `.advice()` method is present in the DSL definition:

```
@Bean
public TcpOutboundGateway tcpOut() {
    TcpOutboundGateway gateway = new TcpOutboundGateway();
    gateway.setConnectionFactory(cf());
    gateway.setAdviceChain(Collections.singletonList(fooAdvice()));
    return gateway;
}

@Bean
public IntegrationFlow clientTcpFlow() {
    return f -> f
        .handle(tcpOut(), e -> e.advice(testAdvice()))
        .transform(Transformers.objectToString());
}
```

That is they are not merged, only the `testAdvice()` bean is used in this case.

11.5. Transformers

The DSL API provides a convenient, fluent `Transformers` factory to be used as inline target object definition within the `.transform()` EIP method. The following example shows how to use it:

```
@Bean
public IntegrationFlow transformFlow() {
    return IntegrationFlows.from("input")
            .transform(Transformers.fromJson(MyPojo.class))
            .transform(Transformers.serializer())
            .get();
}
```

It avoids inconvenient coding using setters and makes the flow definition more straightforward. Note that you can use `Transformers` to declare target `Transformer` instances as `@Bean` instances and, again, use them from `IntegrationFlow` definition as bean methods. Nevertheless, the DSL parser takes care of bean declarations for inline objects, if they are not yet defined as beans.

See Transformers in the Javadoc for more information and supported factory methods.

Also see Lambdas And `Message<?>` Arguments.

11.6. Inbound Channel Adapters

Typically, message flows start from an inbound channel adapter (such as `<int-jdbc:inbound-channel-adapter>`). The adapter is configured with `<poller>`, and it asks a `MessageSource<?>` to periodically produce messages. Java DSL allows for starting `IntegrationFlow` from a `MessageSource<?>`, too. For this purpose, the `IntegrationFlows` builder factory provides an overloaded `IntegrationFlows.from(MessageSource<?> messageSource)` method. You can configure the `MessageSource<?>` as a bean and provide it as an argument for that method. The second parameter of `IntegrationFlows.from()` is a `Consumer<SourcePollingChannelAdapterSpec>` lambda that lets you provide options (such as `PollerMetadata` or `SmartLifecycle`) for the `SourcePollingChannelAdapter`. The following example shows how to use the fluent API and a lambda to create an `IntegrationFlow`:

```
@Bean
public MessageSource<Object> jdbcMessageSource() {
    return new JdbcPollingChannelAdapter(this.dataSource, "SELECT * FROM
something");
}

@Bean
public IntegrationFlow pollingFlow() {
    return IntegrationFlows.from(jdbcMessageSource(),
                c -> c.poller(Pollers.fixedRate(100).maxMessagesPerPoll(1)))
            .transform(Transformers.toJson())
            .channel("furtherProcessChannel")
            .get();
}
```

For those cases that have no requirements to build `Message` objects directly, you can use the `IntegrationFlows.from()` variant that is based on the `java.util.function.Supplier` . The result of the `Supplier.get()` is automatically wrapped in a `Message` (if it is not already a `Message`).

11.7. Message Routers

Spring Integration natively provides specialized router types, including:

- `HeaderValueRouter`
- `PayloadTypeRouter`
- `ExceptionTypeRouter`
- `RecipientListRouter`
- `XPathRouter`

As with many other DSL `IntegrationFlowBuilder` EIP methods, the `route()` method can apply any `AbstractMessageRouter` implementation or, for convenience, a `String` as a SpEL expression or a `ref-method` pair. In addition, you can configure `route()` with a lambda and use a lambda for a `Consumer<RouterSpec<MethodInvokingRouter>>`. The fluent API also provides `AbstractMappingMessageRouter` options such as `channelMapping(String key, String channelName)` pairs, as the following example shows:

```
@Bean
public IntegrationFlow routeFlowByLambda() {
    return IntegrationFlows.from("routerInput")
            .<Integer, Boolean>route(p -> p % 2 == 0,
                    m -> m.suffix("Channel")
                            .channelMapping(true, "even")
                            .channelMapping(false, "odd")
            )
            .get();
}
```

The following example shows a simple expression-based router:

```
@Bean
public IntegrationFlow routeFlowByExpression() {
    return IntegrationFlows.from("routerInput")
            .route("headers['destChannel']")
            .get();
}
```

The `routeToRecipients()` method takes a `Consumer<RecipientListRouterSpec>`, as the following example shows:

```
@Bean
public IntegrationFlow recipientListFlow() {
    return IntegrationFlows.from("recipientListInput")
            .<String, String>transform(p -> p.replaceFirst("Payload", ""))
            .routeToRecipients(r -> r
                    .recipient("thing1-channel", "'thing1' == payload")
                    .recipientMessageSelector("thing2-channel", m ->
                            m.getHeaders().containsKey("recipient")
                                    && (boolean) m.getHeaders().get("recipient"))
                    .recipientFlow("'thing1' == payload or 'thing2' == payload or
'thing3' == payload",
                            f -> f.<String, String>transform(String::toUpperCase)
                                    .channel(c -> c.queue(
"recipientListSubFlow1Result")))
                    .recipientFlow((String p) -> p.startsWith("thing3"),
                            f -> f.transform("Hello "::concat)
                                    .channel(c -> c.queue(
"recipientListSubFlow2Result")))
                    .recipientFlow(new FunctionExpression<Message<?>>(m ->
                                    "thing3".equals(m.getPayload())),
                            f -> f.channel(c -> c.queue(
"recipientListSubFlow3Result")))
                    .defaultOutputToParentFlow())
            .get();
}
```

The `.defaultOutputToParentFlow()` of the `.routeToRecipients()` definition lets you set the router's `defaultOutput` as a gateway to continue a process for the unmatched messages in the main flow.

Also see Lambdas And `Message<?>` Arguments.

11.8. Splitters

To create a splitter, use the `split()` EIP method. By default, if the payload is an `Iterable`, an `Iterator`, an `Array`, a `Stream`, or a reactive `Publisher`, the `split()` method outputs each item as an individual message. It accepts a lambda, a SpEL expression, or any `AbstractMessageSplitter` implementation. Alternatively, you can use it without parameters to provide the `DefaultMessageSplitter`. The following example shows how to use the `split()` method by providing a lambda:

```
@Bean
public IntegrationFlow splitFlow() {
    return IntegrationFlows.from("splitInput")
                .split(s -> s.applySequence(false).delimiters(","))
                .channel(MessageChannels.executor(taskExecutor()))
                .get();
}
```

The preceding example creates a splitter that splits a message containing a comma-delimited `String`.

Also see Lambdas And `Message<?>` Arguments.

11.9. Aggregators and Resequencers

An `Aggregator` is conceptually the opposite of a `Splitter`. It aggregates a sequence of individual messages into a single message and is necessarily more complex. By default, an aggregator returns a message that contains a collection of payloads from incoming messages. The same rules are applied for the `Resequencer`. The following example shows a canonical example of the splitter-aggregator pattern:

```
@Bean
public IntegrationFlow splitAggregateFlow() {
    return IntegrationFlows.from("splitAggregateInput")
            .split()
            .channel(MessageChannels.executor(this.taskExecutor()))
            .resequence()
            .aggregate()
            .get();
}
```

The `split()` method splits the list into individual messages and sends them to the `ExecutorChannel`. The `resequence()` method reorders messages by sequence details found in the message headers. The `aggregate()` method collects those messages.

However, you can change the default behavior by specifying a release strategy and correlation strategy, among other things. Consider the following example:

```
.aggregate(a ->
        a.correlationStrategy(m -> m.getHeaders().get("myCorrelationKey"))
            .releaseStrategy(g -> g.size() > 10)
            .messageStore(messageStore()))
```

The preceding example correlates messages that have `myCorrelationKey` headers and releases the messages once at least ten have been accumulated.

Similar lambda configurations are provided for the `resequence()` EIP method.

11.10. Service Activators and the `.handle()` method

The `.handle()` EIP method's goal is to invoke any `MessageHandler` implementation or any method on some POJO. Another option is to define an "activity" by using lambda expressions. Consequently, we introduced a generic `GenericHandler<P>` functional interface. Its `handle` method requires two arguments: `P payload` and `MessageHeaders headers` (starting with version 5.1). Having that, we can define a flow as follows:

```
@Bean
public IntegrationFlow myFlow() {
    return IntegrationFlows.from("flow3Input")
        .<Integer>handle((p, h) -> p * 2)
        .get();
}
```

The preceding example doubles any integer it receives.

However, one main goal of Spring Integration is `loose coupling`, through runtime type conversion from message payload to the target arguments of the message handler. Since Java does not support generic type resolution for lambda classes, we introduced a workaround with an additional `payloadType` argument for the most EIP methods and `LambdaMessageProcessor`. Doing so delegates the hard conversion work to Spring's `ConversionService`, which uses the provided `type` and the requested message to target method arguments. The following example shows what the resulting `IntegrationFlow` might look like:

```
@Bean
public IntegrationFlow integerFlow() {
    return IntegrationFlows.from("input")
            .<byte[], String>transform(p - > new String(p, "UTF-8"))
            .handle(Integer.class, (p, h) -> p * 2)
            .get();
}
```

We also can register some `BytesToIntegerConverter` within `ConversionService` to get rid of that additional `.transform()`:

```
@Bean
@IntegrationConverter
public BytesToIntegerConverter bytesToIntegerConverter() {
   return new BytesToIntegerConverter();
}

@Bean
public IntegrationFlow integerFlow() {
    return IntegrationFlows.from("input")
              .handle(Integer.class, (p, h) -> p * 2)
             .get();
}
```

Also see Lambdas And Message<?> Arguments.

11.11. Operator log()

For convenience, to log the message journey through the Spring Integration flow (<logging-channel-adapter>), a log() operator is presented. Internally, it is represented by the WireTap ChannelInterceptor with a LoggingHandler as its subscriber. It is responsible for logging the incoming message into the next endpoint or the current channel. The following example shows how to use LoggingHandler:

```
.filter(...)
.log(LoggingHandler.Level.ERROR, "test.category", m -> m.getHeaders().getId())
.route(...)
```

In the preceding example, an id header is logged at the ERROR level onto test.category only for messages that passed the filter and before routing.

When this operator is used at the end of a flow, it is a one-way handler and the flow ends. To make it as a reply-producing flow, you can either use a simple bridge() after the log() or, starting with version 5.1, you can use a logAndReply() operator instead. logAndReply can only be used at the end of a flow.

11.12. Operator intercept()

Starting with version 5.3, the intercept() operator allows to register one or more ChannelInterceptor instances at the current MessageChannel in the flow. This is an alternative to creating an explicit MessageChannel via the MessageChannels API. The following example uses a MessageSelectingInterceptor to reject certain messages with an exception:

```
.transform(...)
.intercept(new MessageSelectingInterceptor(m -> m.getPayload().isValid()))
.handle(...)
```

11.13. MessageChannelSpec.wireTap()

Spring Integration includes a `.wireTap()` fluent API `MessageChannelSpec` builders. The following example shows how to use the `wireTap` method to log input:

```
@Bean
public QueueChannelSpec myChannel() {
    return MessageChannels.queue()
            .wireTap("loggingFlow.input");
}

@Bean
public IntegrationFlow loggingFlow() {
    return f -> f.log();
}
```

If the `MessageChannel` is an instance of `InterceptableChannel`, the `log()`, `wireTap()` or `intercept()` operators are applied to the current `MessageChannel`. Otherwise, an intermediate `DirectChannel` is injected into the flow for the currently configured endpoint. In the following example, the `WireTap` interceptor is added to `myChannel` directly, because `DirectChannel` implements `InterceptableChannel`:

```
@Bean
MessageChannel myChannel() {
    return new DirectChannel();
}

...
    .channel(myChannel())
    .log()
}
```

When the current `MessageChannel` does not implement `InterceptableChannel`, an implicit `DirectChannel` and `BridgeHandler` are injected into the `IntegrationFlow`, and the `WireTap` is added to this new `DirectChannel`. The following example does not have any channel declaration:

```
.handle(...)
.log()
}
```

In the preceding example (and any time no channel has been declared), an implicit `DirectChannel` is injected in the current position of the `IntegrationFlow` and used as an output channel for the currently configured `ServiceActivatingHandler` (from the `.handle()`, described earlier).

11.14. Working With Message Flows

`IntegrationFlowBuilder` provides a top-level API to produce integration components wired to message flows. When your integration may be accomplished with a single flow (which is often the case), this is convenient. Alternately `IntegrationFlow` instances can be joined via `MessageChannel` instances.

By default, `MessageFlow` behaves as a "chain" in Spring Integration parlance. That is, the endpoints are automatically and implicitly wired by `DirectChannel` instances. The message flow is not actually constructed as a chain, which offers much more flexibility. For example, you may send a message to any component within the flow, if you know its `inputChannel` name (that is, if you explicitly define it). You may also reference externally defined channels within a flow to allow the use of channel adapters (to enable remote transport protocols, file I/O, and so on), instead of direct channels. As such, the DSL does not support the Spring Integration `chain` element, because it does not add much value in this case.

Since the Spring Integration Java DSL produces the same bean definition model as any other configuration options and is based on the existing Spring Framework `@Configuration` infrastructure, it can be used together with XML definitions and wired with Spring Integration messaging annotation configuration.

You can also define direct `IntegrationFlow` instances by using a lambda. The following example shows how to do so:

```
@Bean
public IntegrationFlow lambdaFlow() {
    return f -> f.filter("World"::equals)
                 .transform("Hello "::concat)
                 .handle(System.out::println);
}
```

The result of this definition is the same set of integration components that are wired with an implicit direct channel. The only limitation here is that this flow is started with a named direct channel - `lambdaFlow.input`. Also, a Lambda flow cannot start from `MessageSource` or `MessageProducer`.

Starting with version 5.1, this kind of `IntegrationFlow` is wrapped to the proxy to expose lifecycle

control and provide access to the `inputChannel` of the internally associated `StandardIntegrationFlow`.

Starting with version 5.0.6, the generated bean names for the components in an `IntegrationFlow` include the flow bean followed by a dot (`.`) as a prefix. For example, the `ConsumerEndpointFactoryBean` for the `.transform("Hello "::concat)` in the preceding sample results in a bean name of `lambdaFlow.o.s.i.config.ConsumerEndpointFactoryBean#0`. (The `o.s.i` is a shortened from `org.springframework.integration` to fit on the page.) The `Transformer` implementation bean for that endpoint has a bean name of `lambdaFlow.transformer#0` (starting with version 5.1), where instead of a fully qualified name of the `MethodInvokingTransformer` class, its component type is used. The same pattern is applied for all the `NamedComponent` s when the bean name has to be generated within the flow. These generated bean names are prepended with the flow ID for purposes such as parsing logs or grouping components together in some analysis tool, as well as to avoid a race condition when we concurrently register integration flows at runtime. See Dynamic and Runtime Integration Flows for more information.

11.15. `FunctionExpression`

We introduced the `FunctionExpression` class (an implementation of SpEL's `Expression` interface) to let us use lambdas and `generics`. The `Function<T, R>` option is provided for the DSL components, along with an `expression` option, when there is the implicit `Strategy` variant from Core Spring Integration. The following example shows how to use a function expression:

```
.enrich(e -> e.requestChannel("enrichChannel")
              .requestPayload(Message::getPayload)
              .propertyFunction("date", m -> new Date()))
```

The `FunctionExpression` also supports runtime type conversion, as is done in `SpelExpression`.

11.16. Sub-flows support

Some of `if…else` and `publish-subscribe` components provide the ability to specify their logic or mapping by using sub-flows. The simplest sample is `.publishSubscribeChannel()`, as the following example shows:

```
@Bean
public IntegrationFlow subscribersFlow() {
    return flow -> flow
            .publishSubscribeChannel(Executors.newCachedThreadPool(), s -> s
                    .subscribe(f -> f
                            .<Integer>handle((p, h) -> p / 2)
                            .channel(c -> c.queue("subscriber1Results")))
                    .subscribe(f -> f
                            .<Integer>handle((p, h) -> p * 2)
                            .channel(c -> c.queue("subscriber2Results"))))
            .<Integer>handle((p, h) -> p * 3)
            .channel(c -> c.queue("subscriber3Results"));
}
```

You can achieve the same result with separate `IntegrationFlow` `@Bean` definitions, but we hope you find the sub-flow style of logic composition useful. We find that it results in shorter (and so more readable) code.

Starting with version 5.3, a `BroadcastCapableChannel`-based `publishSubscribeChannel()` implementation is provided to configure sub-flow subscribers on broker-backed message channels. For example we now can configure several subscribers as sub-flows on the `Jms.publishSubscribeChannel()`:

```
@Bean
public BroadcastCapableChannel jmsPublishSubscribeChannel() {
    return Jms.publishSubscribeChannel(jmsConnectionFactory())
                .destination("pubsub")
                .get();
}

@Bean
public IntegrationFlow pubSubFlow() {
    return f -> f
            .publishSubscribeChannel(jmsPublishSubscribeChannel(),
                    pubsub -> pubsub
                            .subscribe(subFlow -> subFlow
                                .channel(c -> c.queue("jmsPubSubBridgeChannel1")))
                            .subscribe(subFlow -> subFlow
                                .channel(c -> c.queue("jmsPubSubBridgeChannel2"))
));
}

@Bean
public BroadcastCapableChannel jmsPublishSubscribeChannel(ConnectionFactory
jmsConnectionFactory) {
    return (BroadcastCapableChannel) Jms.publishSubscribeChannel
(jmsConnectionFactory)
            .destination("pubsub")
            .get();
}
```

A similar `publish-subscribe` sub-flow composition provides the `.routeToRecipients()` method.

Another example is using `.discardFlow()` instead of `.discardChannel()` on the `.filter()` method.

The `.route()` deserves special attention. Consider the following example:

```
@Bean
public IntegrationFlow routeFlow() {
    return f -> f
            .<Integer, Boolean>route(p -> p % 2 == 0,
                    m -> m.channelMapping("true", "evenChannel")
                            .subFlowMapping("false", sf ->
                                    sf.<Integer>handle((p, h) -> p * 3)))
            .transform(Object::toString)
            .channel(c -> c.queue("oddChannel"));
}
```

The `.channelMapping()` continues to work as it does in regular `Router` mapping, but the `.subFlowMapping()` tied that sub-flow to the main flow. In other words, any router's sub-flow returns to the main flow after `.route()`.

Sometimes, you need to refer to an existing `IntegrationFlow @Bean` from the `.subFlowMapping()`. The following example shows how to do so:

```
@Bean
public IntegrationFlow splitRouteAggregate() {
    return f -> f
            .split()
            .<Integer, Boolean>route(o -> o % 2 == 0,
                    m -> m
                            .subFlowMapping(true, oddFlow())
                            .subFlowMapping(false, sf -> sf.gateway
(evenFlow())))
            .aggregate();
}

@Bean
public IntegrationFlow oddFlow() {
    return f -> f.handle(m -> System.out.println("odd"));
}

@Bean
public IntegrationFlow evenFlow() {
    return f -> f.handle((p, h) -> "even");
}
```

In this case, when you need to receive a reply from such a sub-flow and continue the main flow, this `IntegrationFlow` bean reference (or its input channel) has to be wrapped with a `.gateway()` as shown in the preceding example. The `oddFlow()` reference in the preceding example is not wrapped to the `.gateway()`. Therefore, we do not expect a reply from this routing branch. Otherwise, you end up with an exception similar to the following:

```
Caused by: org.springframework.beans.factory.BeanCreationException:
    The 'currentComponent'
(org.springframework.integration.router.MethodInvokingRouter@7965a51c)
    is a one-way 'MessageHandler' and it isn't appropriate to configure
'outputChannel'.
    This is the end of the integration flow.
```

When you configure a sub-flow as a lambda, the framework handles the request-reply interaction with the sub-flow and a gateway is not needed.

Sub-flows can be nested to any depth, but we do not recommend doing so. In fact, even in the

router case, adding complex sub-flows within a flow would quickly begin to look like a plate of spaghetti and be difficult for a human to parse.

In cases where the DSL supports a subflow configuration, when a channel is normally needed for the component being configured, and that subflow starts with a `channel()` element, the framework implicitly places a `bridge()` between the component output channel and the flow's input channel. For example, in this `filter` definition:

```
.filter(p -> p instanceof String, e -> e
    .discardFlow(df -> df
                        .channel(MessageChannels.queue())
                        ...)
```

the Framework internally creates a `DirectChannel` bean for injecting into the `MessageFilter.discardChannel`. Then it wraps the subflow into an `IntegrationFlow` starting with this implicit channel for the subscription and places a `bridge` before the `channel()` specified in the flow. When an existing `IntegrationFlow` bean is used as a subflow reference (instead of an inline subflow, e.g. a lambda), there is no such bridge required because the framework can resolve the first channel from the flow bean. With an inline subflow, the input channel is not yet available.

11.17. Using Protocol Adapters

All of the examples shown so far illustrate how the DSL supports a messaging architecture by using the Spring Integration programming model. However, we have yet to do any real integration. Doing so requires access to remote resources over HTTP, JMS, AMQP, TCP, JDBC, FTP, SMTP, and so on or access to the local file system. Spring Integration supports all of these and more. Ideally, the DSL should offer first class support for all of them, but it is a daunting task to implement all of these and keep up as new adapters are added to Spring Integration. So the expectation is that the DSL is continually catching up with Spring Integration.

Consequently, we provide the high-level API to seamlessly define protocol-specific messaging. We do so with the factory and builder patterns and with lambdas. You can think of the factory classes as "Namespace Factories", because they play the same role as the XML namespace for components from the concrete protocol-specific Spring Integration modules. Currently, Spring Integration Java DSL supports the `Amqp`, `Feed`, `Jms`, `Files`, `(S)Ftp`, `Http`, `JPA`, `MongoDb`, `TCP/UDP`, `Mail`, `WebFlux`, and `Scripts` namespace factories. The following example shows how to use three of them (`Amqp`, `Jms`, and `Mail`):

```
@Bean
public IntegrationFlow amqpFlow() {
    return IntegrationFlows.from(Amqp.inboundGateway(this.rabbitConnectionFactory,
queue()))
            .transform("hello "::concat)
            .transform(String.class, String::toUpperCase)
            .get();
}

@Bean
public IntegrationFlow jmsOutboundGatewayFlow() {
    return IntegrationFlows.from("jmsOutboundGatewayChannel")
            .handle(Jms.outboundGateway(this.jmsConnectionFactory)
                        .replyContainer(c ->
                                    c.concurrentConsumers(3)
                                            .sessionTransacted(true))
                        .requestDestination("jmsPipelineTest"))
            .get();
}

@Bean
public IntegrationFlow sendMailFlow() {
    return IntegrationFlows.from("sendMailChannel")
            .handle(Mail.outboundAdapter("localhost")
                            .port(smtpPort)
                            .credentials("user", "pw")
                            .protocol("smtp")
                            .javaMailProperties(p -> p.put("mail.debug", "true")),
                    e -> e.id("sendMailEndpoint"))
            .get();
}
```

The preceding example shows how to use the “namespace factories” as inline adapters declarations. However, you can use them from `@Bean` definitions to make the `IntegrationFlow` method chain more readable.

We are soliciting community feedback on these namespace factories before we spend effort on others. We also appreciate any input into prioritization for which adapters and gateways we should support next.

You can find more Java DSL samples in the protocol-specific chapters throughout this reference manual.

All other protocol channel adapters may be configured as generic beans and wired to the `IntegrationFlow`, as the following examples show:

```
@Bean
public QueueChannelSpec wrongMessagesChannel() {
    return MessageChannels
            .queue()
            .wireTap("wrongMessagesWireTapChannel");
}

@Bean
public IntegrationFlow xpathFlow(MessageChannel wrongMessagesChannel) {
    return IntegrationFlows.from("inputChannel")
            .filter(new StringValueTestXPathMessageSelector("namespace-uri(/*)",
"my:namespace"),
                    e -> e.discardChannel(wrongMessagesChannel))
            .log(LoggingHandler.Level.ERROR, "test.category", m -> m.getHeaders()
.getId())
            .route(xpathRouter(wrongMessagesChannel))
            .get();
}

@Bean
public AbstractMappingMessageRouter xpathRouter(MessageChannel
wrongMessagesChannel) {
    XPathRouter router = new XPathRouter("local-name(/*)");
    router.setEvaluateAsString(true);
    router.setResolutionRequired(false);
    router.setDefaultOutputChannel(wrongMessagesChannel);
    router.setChannelMapping("Tags", "splittingChannel");
    router.setChannelMapping("Tag", "receivedChannel");
    return router;
}
```

11.18. IntegrationFlowAdapter

The `IntegrationFlow` interface can be implemented directly and specified as a component for scanning, as the following example shows:

```
@Component
public class MyFlow implements IntegrationFlow {

    @Override
    public void configure(IntegrationFlowDefinition<?> f) {
        f.<String, String>transform(String::toUpperCase);
    }

}
```

It is picked up by the `IntegrationFlowBeanPostProcessor` and correctly parsed and registered in the application context.

For convenience and to gain the benefits of loosely coupled architecture, we provide the `IntegrationFlowAdapter` base class implementation. It requires a `buildFlow()` method implementation to produce an `IntegrationFlowDefinition` by using one of `from()` methods, as the following example shows:

```
@Component
public class MyFlowAdapter extends IntegrationFlowAdapter {

    private final AtomicBoolean invoked = new AtomicBoolean();

    public Date nextExecutionTime(TriggerContext triggerContext) {
          return this.invoked.getAndSet(true) ? null : new Date();
    }

    @Override
    protected IntegrationFlowDefinition<?> buildFlow() {
        return from(this::messageSource,
                      e -> e.poller(p -> p.trigger(this::nextExecutionTime)))
                .split(this)
                .transform(this)
                .aggregate(a -> a.processor(this, null), null)
                .enrichHeaders(Collections.singletonMap("thing1", "THING1"))
                .filter(this)
                .handle(this)
                .channel(c -> c.queue("myFlowAdapterOutput"));
    }

    public String messageSource() {
         return "T,H,I,N,G,2";
    }

    @Splitter
    public String[] split(String payload) {
         return StringUtils.commaDelimitedListToStringArray(payload);
    }

    @Transformer
    public String transform(String payload) {
         return payload.toLowerCase();
    }

    @Aggregator
    public String aggregate(List<String> payloads) {
           return payloads.stream().collect(Collectors.joining());
    }

    @Filter
    public boolean filter(@Header Optional<String> thing1) {
            return thing1.isPresent();
    }

    @ServiceActivator
    public String handle(String payload, @Header String thing1) {
           return payload + ":" + thing1;
```

```
    }

}
```

11.19. Dynamic and Runtime Integration Flows

`IntegrationFlow` and all its dependent components can be registered at runtime. Before version 5.0, we used the `BeanFactory.registerSingleton()` hook. Starting in the Spring Framework `5.0`, we use the `instanceSupplier` hook for programmatic `BeanDefinition` registration. The following example shows how to programmatically register a bean:

```
BeanDefinition beanDefinition =
         BeanDefinitionBuilder.genericBeanDefinition((Class<Object>) bean.
getClass(), () -> bean)
                .getRawBeanDefinition();

((BeanDefinitionRegistry) this.beanFactory).registerBeanDefinition(beanName,
beanDefinition);
```

Note that, in the preceding example, the `instanceSupplier` hook is the last parameter to the `genericBeanDefinition` method, provided by a lambda in this case.

All the necessary bean initialization and lifecycle is done automatically, as it is with the standard context configuration bean definitions.

To simplify the development experience, Spring Integration introduced `IntegrationFlowContext` to register and manage `IntegrationFlow` instances at runtime, as the following example shows:

```
@Autowired
private AbstractServerConnectionFactory server1;

@Autowired
private IntegrationFlowContext flowContext;

...

@Test
public void testTcpGateways() {
    TestingUtilities.waitListening(this.server1, null);

    IntegrationFlow flow = f -> f
            .handle(Tcp.outboundGateway(Tcp.netClient("localhost", this.server1
.getPort())
                    .serializer(TcpCodecs.crlf())
                    .deserializer(TcpCodecs.lengthHeader1())
                    .id("client1"))
                .remoteTimeout(m -> 5000))
            .transform(Transformers.objectToString());

    IntegrationFlowRegistration theFlow = this.flowContext.registration(flow)
.register();
    assertThat(theFlow.getMessagingTemplate().convertSendAndReceive("foo", String
.class), equalTo("FOO"));
}
```

This is useful when we have multiple configuration options and have to create several instances of similar flows. To do so, we can iterate our options and create and register `IntegrationFlow` instances within a loop. Another variant is when our source of data is not Spring-based and we must create it on the fly. Such a sample is Reactive Streams event source, as the following example shows:

```
Flux<Message<?>> messageFlux =
    Flux.just("1,2,3,4")
        .map(v -> v.split(","))
        .flatMapIterable(Arrays::asList)
        .map(Integer::parseInt)
        .map(GenericMessage<Integer>::new);

QueueChannel resultChannel = new QueueChannel();

IntegrationFlow integrationFlow =
    IntegrationFlows.from(messageFlux)
        .<Integer, Integer>transform(p -> p * 2)
        .channel(resultChannel)
        .get();

this.integrationFlowContext.registration(integrationFlow)
            .register();
```

The `IntegrationFlowRegistrationBuilder` (as a result of the `IntegrationFlowContext.registration()`) can be used to specify a bean name for the `IntegrationFlow` to register, to control its `autoStartup`, and to register, non-Spring Integration beans. Usually, those additional beans are connection factories (AMQP, JMS, (S)FTP, TCP/UDP, and others.), serializers and deserializers, or any other required support components.

You can use the `IntegrationFlowRegistration.destroy()` callback to remove a dynamically registered `IntegrationFlow` and all its dependent beans when you no longer need them. See the `IntegrationFlowContext` Javadoc for more information.

Starting with version 5.0.6, all generated bean names in an `IntegrationFlow` definition are prepended with the flow ID as a prefix. We recommend always specifying an explicit flow ID. Otherwise, a synchronization barrier is initiated in the `IntegrationFlowContext`, to generate the bean name for the `IntegrationFlow` and register its beans. We synchronize on these two operations to avoid a race condition when the same generated bean name may be used for different `IntegrationFlow` instances.

Also, starting with version 5.0.6, the registration builder API has a new method: `useFlowIdAsPrefix()`. This is useful if you wish to declare multiple instances of the same flow and avoid bean name collisions when components in the flows have the same ID, as the following example shows:

```
private void registerFlows() {
    IntegrationFlowRegistration flow1 =
              this.flowContext.registration(buildFlow(1234))
                    .id("tcp1")
                    .useFlowIdAsPrefix()
                    .register();

    IntegrationFlowRegistration flow2 =
              this.flowContext.registration(buildFlow(1235))
                    .id("tcp2")
                    .useFlowIdAsPrefix()
                    .register();
}

private IntegrationFlow buildFlow(int port) {
    return f -> f
            .handle(Tcp.outboundGateway(Tcp.netClient("localhost", port)
                    .serializer(TcpCodecs.crlf())
                    .deserializer(TcpCodecs.lengthHeader1())
                    .id("client"))
               .remoteTimeout(m -> 5000))
            .transform(Transformers.objectToString());
}
```

In this case, the message handler for the first flow can be referenced with bean a name of `tcp1.client.handler`.

An `id` attribute is required when you usE `useFlowIdAsPrefix()`.

11.20. `IntegrationFlow` as a Gateway

The `IntegrationFlow` can start from the service interface that provides a `GatewayProxyFactoryBean` component, as the following example shows:

```
public interface ControlBusGateway {

    void send(String command);
}

...

@Bean
public IntegrationFlow controlBusFlow() {
    return IntegrationFlows.from(ControlBusGateway.class)
            .controlBus()
            .get();
}
```

All the proxy for interface methods are supplied with the channel to send messages to the next integration component in the `IntegrationFlow`. You can mark the service interface with the `@MessagingGateway` annotation and mark the methods with the `@Gateway` annotations. Nevertheless, the `requestChannel` is ignored and overridden with that internal channel for the next component in the `IntegrationFlow`. Otherwise, creating such a configuration by using `IntegrationFlow` does not make sense.

By default a `GatewayProxyFactoryBean` gets a conventional bean name, such as `[FLOW_BEAN_NAME.gateway]`. You can change that ID by using the `@MessagingGateway.name()` attribute or the overloaded `IntegrationFlows.from(Class<?> serviceInterface, Consumer<GatewayProxySpec> endpointConfigurer)` factory method. Also all the attributes from the `@MessagingGateway` annotation on the interface are applied to the target `GatewayProxyFactoryBean`. When annotation configuration is not applicable, the `Consumer<GatewayProxySpec>` variant can be used for providing appropriate option for the target proxy. This DSL method is available starting with version 5.2.

With Java 8, you can even create an integration gateway with the `java.util.function` interfaces, as the following example shows:

```
@Bean
public IntegrationFlow errorRecovererFlow() {
    return IntegrationFlows.from(Function.class, (gateway) -> gateway.beanName(
"errorRecovererFunction"))
            .handle((GenericHandler<?>) (p, h) -> {
                throw new RuntimeException("intentional");
            }, e -> e.advice(retryAdvice()))
            .get();
}
```

That `errorRecovererFlow` can be used as follows:

```
@Autowired
@Qualifier("errorRecovererFunction")
private Function<String, String> errorRecovererFlowGateway;
```

11.21. DSL Extensions

Starting with version 5.3, an `IntegrationFlowExtension` has been introduced to allow extension of the existing Java DSL with custom or composed EIP-operators. All that is needed is an extension of this class that provides methods which can be used in the `IntegrationFlow` bean definitions. The extension class can also be used for custom `IntegrationComponentSpec` configuration; for example, missed or default options can be implemented in the existing `IntegrationComponentSpec` extension. The sample below demonstrates a composite custom operator and usage of an `AggregatorSpec` extension for a default custom `outputProcessor`:

```
public class CustomIntegrationFlowDefinition
        extends IntegrationFlowExtension<CustomIntegrationFlowDefinition> {

    public CustomIntegrationFlowDefinition upperCaseAfterSplit() {
        return split()
                .transform("payload.toUpperCase()");
    }

    public CustomIntegrationFlowDefinition customAggregate(Consumer
<CustomAggregatorSpec> aggregator) {
        return register(new CustomAggregatorSpec(), aggregator);
    }

}

public class CustomAggregatorSpec extends AggregatorSpec {

    CustomAggregatorSpec() {
        outputProcessor(group ->
                group.getMessages()
                        .stream()
                        .map(Message::getPayload)
                        .map(String.class::cast)
                        .collect(Collectors.joining(", ")));
    }

}
```

For a method chain flow the new DSL operator in these extensions must return the extension class. This way a target `IntegrationFlow` definition will work with new and existing DSL operators:

```
@Bean
public IntegrationFlow customFlowDefinition() {
    return
            new CustomIntegrationFlowDefinition()
                    .log()
                    .upperCaseAfterSplit()
                    .channel("innerChannel")
                    .customAggregate(customAggregatorSpec ->
                            customAggregatorSpec.expireGroupsUponCompletion(true))
                    .logAndReply();
}
```

Chapter 12. Kotlin DSL

The Kotlin DSL is a wrapper and extension to Java DSL and aimed to make Spring Integration development on Kotlin as smooth and straightforward as possible with interoperability with the existing Java API and Kotlin language-specific structures.

All you need to get started is just an import for `org.springframework.integration.dsl.integrationFlow` - an overloaded global function for Kotlin DSL.

For `IntegrationFlow` definitions as lambdas we typically don't need anything else from Kotlin and just declare a bean like this:

```
@Bean
fun oddFlow() =
IntegrationFlow { flow ->
    flow.handle<Any> { _, _ -> "odd" }
}
```

In this case Kotlin understands that the lambda should be translated into `IntegrationFlow` anonymous instance and target Java DSL processor parses this construction properly into Java objects.

As an alternative to the construction above and for consistency with use-cases explained below, a Kotlin-specif DSL should be used for declaring integration flows in the **builder** pattern style:

```
@Bean
fun flowLambda() =
    integrationFlow {
        filter<String> { it === "test" }
        wireTap {
                    handle { println(it.payload) }
                }
        transform<String, String> { it.toUpperCase() }
    }
```

Such a global `integrationFlow()` function expects a lambda in builder style for a `KotlinIntegrationFlowDefinition` (a Kotlin wrapper for the `IntegrationFlowDefinition`) and produces a regular `IntegrationFlow` lambda implementation. See more overloaded `integrationFlow()` variants below.

Many other scenarios require an `IntegrationFlow` to be started from source of data (e.g. `JdbcPollingChannelAdapter`, `JmsInboundGateway` or just an existing `MessageChannel`). For this purpose, the Spring Integration Java DSL provides an `IntegrationFlows` factory with its large number of

overloaded `from()` methods. This factory can be used in Kotlin as well:

```
@Bean
fun flowFromSupplier() =
         IntegrationFlows.from<String>({ "bar" }) { e -> e.poller { p ->
p.fixedDelay(10).maxMessagesPerPoll(1) } }
                 .channel { c -> c.queue("fromSupplierQueue") }
                 .get()
```

But unfortunately not all `from()` methods are compatible with Kotlin structures. To fix the gap, this project provides a Kotlin DSL around an `IntegrationFlows` factory. It is implemented as a set of overloaded `integrationFlow()` functions. With a consumer for a `KotlinIntegrationFlowDefinition` to declare the rest of the flow as an `IntegrationFlow` lambda to reuse the mentioned above experience and also avoid `get()` call in the end. For example:

```
@Bean
fun functionFlow() =
        integrationFlow<Function<String, String>>({ beanName("functionGateway") })
{
            transform<String, String> { it.toUpperCase() }
        }

@Bean
fun messageSourceFlow() =
        integrationFlow(MessageProcessorMessageSource { "testSource" },
                { poller { it.fixedDelay(10).maxMessagesPerPoll(1) } }) {
            channel { queue("fromSupplierQueue") }
        }
```

In addition, Kotlin extensions are provided for the Java DSL API which needs some refinement for Kotlin structures. For example `IntegrationFlowDefinition<*>` requires a reifying for many methods with `Class<P>` argument:

```
@Bean
fun convertFlow() =
    integrationFlow("convertFlowInput") {
        convert<TestPojo>()
    }
```

Chapter 13. System Management

13.1. Metrics and Management

This section describes how to capture metrics for Spring Integration. In recent versions, we have relied more on Micrometer (see micrometer.io), and we plan to use Micrometer even more in future releases.

13.1.1. Configuring Metrics Capture

Prior to version 4.2, metrics were only available when JMX was enabled. See JMX Support.

To enable `MessageSource`, `MessageChannel`, and `MessageHandler` metrics, add an `<int:management/>` bean to the application context (in XML) or annotate one of your `@Configuration` classes with `@EnableIntegrationManagement` (in Java). `MessageSource` instances maintain only counts, `MessageChannel` instances and `MessageHandler` instances maintain duration statistics in addition to counts. See `MessageChannel` Metric Features and MessageHandler Metric Features, later in this chapter.

Doing so causes the automatic registration of the `IntegrationManagementConfigurer` bean in the application context. Only one such bean can exist in the context, and, if registered manually via a `<bean/>` definition, it must have the bean name set to `integrationManagementConfigurer`. This bean applies its configuration to beans after all beans in the context have been instantiated.

In addition to metrics, you can control debug logging in the main message flow. In very high volume applications, even calls to `isDebugEnabled()` can be quite expensive with some logging subsystems. You can disable all such logging to avoid this overhead. Exception logging (debug or otherwise) is not affected by this setting.

The following listing shows the available options for controlling logging:

```
<int:management
    default-logging-enabled="true" ①
    default-counts-enabled="false" ②
    default-stats-enabled="false" ③
    counts-enabled-patterns="foo, !baz, ba*" ④
    stats-enabled-patterns="fiz, buz" ⑤
    metrics-factory="myMetricsFactory" /> ⑥
```

```
@Configuration
@EnableIntegration
@EnableIntegrationManagement(
    defaultLoggingEnabled = "true", ①
    defaultCountsEnabled = "false", ②
    defaultStatsEnabled = "false", ③
    countsEnabled = { "foo", "${count.patterns}" }, ④
    statsEnabled = { "qux", "!*" }, ⑤
    MetricsFactory = "myMetricsFactory") ⑥
public static class ContextConfiguration {
...
}
```

① Set to `false` to disable all logging in the main message flow, regardless of the log system category settings. Set to 'true' to enable debug logging (if also enabled by the logging subsystem). Only applied if you have not explicitly configured the setting in a bean definition. The default is `true`.

② Enable or disable count metrics for components that do not match one of the patterns in <4>. Only applied if you have not explicitly configured the setting in a bean definition. The default is `false`.

③ Enable or disable statistical metrics for components that do not match one of the patterns in <5>. Only applied if you have not explicitly configured the setting in a bean definition. The default is 'false'.

④ A comma-delimited list of patterns for beans for which counts should be enabled. You can negate the pattern with `!`. First match (positive or negative) wins. In the unlikely event that you have a bean name starting with `!`, escape the `!` in the pattern. For example, `\!something` positively matches a bean named `!something`.

⑤ A comma-delimited list of patterns for beans for which statistical metrics should be enabled. You can negate the pattern\ with `!`. First match (positive or negative) wins. In the unlikely event that you have a bean name starting with `!`, escape the `!` in the pattern. `\!something` positively matches a bean named `!something`. The collection of statistics implies the collection of counts.

⑥ A reference to a `MetricsFactory`. See Metrics Factory.

At runtime, counts and statistics can be obtained by calling `getChannelMetrics`, `getHandlerMetrics` and `getSourceMetrics` (all from the `IntegrationManagementConfigurer` class), which return `MessageChannelMetrics`, `MessageHandlerMetrics`, and `MessageSourceMetrics`, respectively.

See the Javadoc for complete information about these classes.

When JMX is enabled (see JMX Support), `IntegrationMBeanExporter` also exposes these metrics.

IMPORTANT: `defaultLoggingEnabled`, `defaultCountsEnabled`, and `defaultStatsEnabled` are applied only if you have not explicitly configured the corresponding setting in a bean definition.

Starting with version 5.0.2, the framework automatically detects whether the application context has a single `MetricsFactory` bean and, if so, uses it instead of the default metrics factory.

These legacy metrics have been deprecated in favor of Micrometer metrics discussed below. Legacy metrics support will be removed in a future release.

13.1.2. Micrometer Integration

Starting with version 5.0.3, the presence of a Micrometer `MeterRegistry` in the application context triggers support for Micrometer metrics in addition to the built-in metrics (note that the legacy built-in metrics will be removed in a future release).

Micrometer was first supported in version 5.0.2, but changes were made to the Micrometer `Meters` in version 5.0.3 to make them more suitable for use in dimensional systems. Further changes were made in 5.0.4. If you use Micrometer, a minimum of version 5.0.4 is recommended, since some of the changes in 5.0.4 were breaking API changes.

To use Micrometer, add one of the `MeterRegistry` beans to the application context. If the `IntegrationManagementConfigurer` detects exactly one `MeterRegistry` bean, it configures a `MicrometerMetricsCaptor` bean with a name of `integrationMicrometerMetricsCaptor`.

For each `MessageHandler` and `MessageChannel`, timers are registered. For each `MessageSource`, a counter is registered.

This only applies to objects that extend `AbstractMessageHandler`, `AbstractMessageChannel`, and `AbstractMessageSource` (which is the case for most framework components).

With Micrometer metrics, the `statsEnabled` flag has no effect, since statistics capture is delegated to Micrometer. The `countsEnabled` flag controls whether the Micrometer `Meter` instances are updated when processing each message.

The `Timer` Meters for send operations on message channels have the following names or tags:

- `name`: `spring.integration.send`
- `tag`: `type:channel`
- `tag`: `name:<componentName>`
- `tag`: `result:(success|failure)`
- `tag`: `exception:(none|exception simple class name)`
- `description`: `Send processing time`

(A `failure` result with a `none` exception means the channel's `send()` operation returned `false`.)

The `Counter` Meters for receive operations on pollable message channels have the following names or tags:

- `name`: `spring.integration.receive`
- `tag`: `type:channel`
- `tag`: `name:<componentName>`
- `tag`: `result:(success|failure)`
- `tag`: `exception:(none|exception simple class name)`
- `description`: `Messages received`

The `Timer` Meters for operations on message handlers have the following names or tags:

- `name`: `spring.integration.send`
- `tag`: `type:handler`
- `tag`: `name:<componentName>`
- `tag`: `result:(success|failure)`
- `tag`: `exception:(none|exception simple class name)`
- `description`: `Send processing time`

The `Counter` meters for message sources have the following names/tags:

- `name`: `spring.integration.receive`
- `tag`: `type:source`
- `tag`: `name:<componentName>`
- `tag`: `result:success`
- `tag`: `exception:none`
- `description`: `Messages received`

In addition, there are three `Gauge` Meters:

- `spring.integration.channels`: The number of `MessageChannels` in the application.
- `spring.integration.handlers`: The number of `MessageHandlers` in the application.
- `spring.integration.sources`: The number of `MessageSources` in the application.

It is possible to customize the names and tags of `Meters` created by integration components by providing a subclass of `MicrometerMetricsCaptor`. The MicrometerCustomMetricsTests test case shows a simple example of how to do that. You can also further customize the meters by overloading the `build()` methods on builder subclasses.

13.1.3. `MessageChannel` Metric Features

These legacy metrics will be removed in a future release. See Micrometer Integration.

Message channels report metrics according to their concrete type. If you are looking at a `DirectChannel`, you see statistics for the send operation. If it is a `QueueChannel`, you also see statistics for the receive operation as well as the count of messages that are currently buffered by this `QueueChannel`. In both cases, some metrics are simple counters (message count and error count), and some are estimates of averages of interesting quantities. The algorithms used to calculate these estimates are described briefly in the following table.

Table 6. MessageChannel Metrics

Metric Type	Example	Algorithm
Count	Send Count	Simple incrementer. Increases by one when an event occurs.
Error Count	Send Error Count	Simple incrementer. Increases by one when an send results in an error.
Duration	Send Duration (method execution time in milliseconds)	Exponential moving average with decay factor (ten by default). Average of the method execution time over roughly the last ten (by default) measurements.
Rate	Send Rate (number of operations per second)	Inverse of Exponential moving average of the interval between events with decay in time (lapsing over 60 seconds by default) and per measurement (last ten events by default).
Error Rate	Send Error Rate (number of errors per second)	Inverse of exponential moving average of the interval between error events with decay in time (lapsing over 60 seconds by default) and per measurement (last ten events by default).
Ratio	Send Success Ratio (ratio of successful to total sends)	Estimate the success ratio as the exponential moving average of the series composed of values (1 for success and 0 for failure, decaying as per the rate measurement over time and events by default). The error ratio is: 1 - success ratio.

13.1.4. MessageHandler Metric Features

These legacy metrics will be removed in a future release. See Micrometer Integration.

The following table shows the statistics maintained for message handlers. Some metrics are simple counters (message count and error count), and one is an estimate of averages of send duration. The algorithms used to calculate these estimates are described briefly in the following table:

Table 7. MessageHandlerMetrics

Metric Type	Example	Algorithm
Count	Handle Count	Simple incrementer. Increases by one when an event occurs.
Error Count	Handler Error Count	Simple incrementer. Increases by one when an invocation results in an error.
Active Count	Handler Active Count	Indicates the number of currently active threads currently invoking the handler (or any downstream synchronous flow).
Duration	Handle Duration (method execution time in milliseconds)	Exponential moving average with decay factor (ten by default). Average of the method execution time over roughly the last ten (default) measurements.

13.1.5. Time-Based Average Estimates

A feature of the time-based average estimates is that they decay with time if no new measurements arrive. To help interpret the behavior over time, the time (in seconds) since the last measurement is also exposed as a metric.

There are two basic exponential models: decay per measurement (appropriate for duration and anything where the number of measurements is part of the metric) and decay per time unit (more suitable for rate measurements where the time in between measurements is part of the metric). Both models depend on the fact that `S(n) = sum(i=0,i=n) w(i) x(i)` has a special form when `w(i) = r^i`, with `r=constant`: `S(n) = x(n) + r S(n-1)` (so you only have to store `S(n-1)` (not the whole series `x(i)`) to generate a new metric estimate from the last measurement). The algorithms used in the duration metrics use `r=exp(-1/M)` with `M=10`. The net effect is that the estimate, `S(n)`, is more heavily weighted to recent measurements and is composed roughly of the last `M` measurements. So `M` is the "window" or lapse rate of the estimate. For the vanilla moving average, `i` is a counter over the number of measurements. For the rate, we interpret `i` as the elapsed time or a combination of elapsed time and a counter (so the metric estimate contains contributions roughly from the last `M` measurements and the last `T` seconds).

13.1.6. Metrics Factory

A strategy interface `MetricsFactory` has been introduced to let you provide custom channel metrics for your `MessageChannel` instances and `MessageHandler` instances. By default, a `DefaultMetricsFactory` provides a default implementation of `MessageChannelMetrics` and `MessageHandlerMetrics`, described earlier. To override the default `MetricsFactory`, configure it as described earlier, by providing a reference to your `MetricsFactory` bean instance. You can either customize the default implementations, as described in the next section, or provide completely different implementations by extending `AbstractMessageChannelMetrics` or `AbstractMessageHandlerMetrics`.

See also Micrometer Integration.

In addition to the default metrics factory described earlier, the framework provides the `AggregatingMetricsFactory`. This factory creates `AggregatingMessageChannelMetrics` and `AggregatingMessageHandlerMetrics` instances. In very high volume scenarios, the cost of capturing

statistics can be prohibitive (the time to make two calls to the system and store the data for each message). The aggregating metrics aggregate the response time over a sample of messages. This can save significant CPU time.

The statistics are likely to be skewed if messages arrive in bursts. These metrics are intended for use with high, constant-volume, message rates.

The following example shows how to define an aggregrating metrics factory:

```
<bean id="aggregatingMetricsFactory"
            class=
"org.springframework.integration.support.management.AggregatingMetricsFactory">
    <constructor-arg value="1000" /> <!-- sample size -->
</bean>
```

The preceding configuration aggregates the duration over 1000 messages. Counts (send and error) are maintained per-message, but the statistics are per 1000 messages.

Customizing the Default Channel and Handler Statistics

See Time-Based Average Estimates and the Javadoc for the `ExponentialMovingAverage*` classes for more information about these values.

By default, the `DefaultMessageChannelMetrics` and `DefaultMessageHandlerMetrics` use a “window” of ten measurements, a rate period of one second (meaning rate per second) and a decay lapse period of one minute.

If you wish to override these defaults, you can provide a custom `MetricsFactory` that returns appropriately configured metrics and provide a reference to it in the MBean exporter, as described earlier.

The following example shows how to do so:

```
public static class CustomMetrics implements MetricsFactory {

    @Override
    public AbstractMessageChannelMetrics createChannelMetrics(String name) {
        return new DefaultMessageChannelMetrics(name,
                new ExponentialMovingAverage(20, 1000000.),
                new ExponentialMovingAverageRate(2000, 120000, 30, true),
                new ExponentialMovingAverageRatio(130000, 40, true),
                new ExponentialMovingAverageRate(3000, 140000, 50, true));
    }

    @Override
    public AbstractMessageHandlerMetrics createHandlerMetrics(String name) {
        return new DefaultMessageHandlerMetrics(name, new
ExponentialMovingAverage(20, 1000000.));
    }

}
```

Advanced Customization

The customizations described earlier are wholesale and apply to all appropriate beans exported by the MBean exporter. This is the extent of customization available when you use XML configuration.

Individual beans can be provided with different implementations using by Java `@Configuration` or programmatically at runtime (after the application context has been refreshed) by invoking the `configureMetrics` methods on `AbstractMessageChannel` and `AbstractMessageHandler`.

Performance Improvement

Previously, the time-based metrics (see Time-Based Average Estimates) were calculated in real time. The statistics are now calculated when retrieved instead. This resulted in a significant performance improvement, at the expense of a small amount of additional memory for each statistic. As discussed earlier, you can disable the statistics altogether while retaining the MBean that allows the invocation of `Lifecycle` methods.

13.2. JMX Support

Spring Integration provides channel Adapters for receiving and publishing JMX Notifications.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-jmx</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-jmx:5.3.0.RC1"
```

An inbound channel adapter allows for polling JMX MBean attribute values, and an outbound channel adapter allows for invoking JMX MBean operations.

13.2.1. Notification-listening Channel Adapter

The notification-listening channel adapter requires a JMX `ObjectName` for the MBean that publishes notifications to which this listener should be registered. A very simple configuration might resemble the following:

```
<int-jmx:notification-listening-channel-adapter id="adapter"
    channel="channel"
    object-name="example.domain:name=publisher"/>
```

The `notification-listening-channel-adapter` registers with an `MBeanServer` at startup, and the default bean name is `mbeanServer`, which happens to be the same bean name generated when using Spring's `<context:mbean-server/>` element. If you need to use a different name, be sure to include the `mbean-server` attribute.

The adapter can also accept a reference to a `NotificationFilter` and a "handback" object to provide some context that is passed back with each notification. Both of those attributes are optional. Extending the preceding example to include those attributes as well as an explicit `MBeanServer` bean name produces the following example:

```
<int-jmx:notification-listening-channel-adapter id="adapter"
    channel="channel"
    mbean-server="someServer"
    object-name="example.domain:name=somePublisher"
    notification-filter="notificationFilter"
    handback="myHandback"/>
```

The _Notification-listening channel adapter is event-driven and registered with the `MBeanServer` directly. It does not require any poller configuration.

For this component only, the `object-name` attribute can contain an object name pattern (for example, "org.something:type=MyType,name=*"). In that case, the adapter receives notifications from all MBeans with object names that match the pattern. In addition, the `object-name` attribute can contain a SpEL reference to a `<util:list>` of object name patterns, as the following example shows:

```
<jmx:notification-listening-channel-adapter id=
"manyNotificationsAdapter"
    channel="manyNotificationsChannel"
    object-name="#{patterns}"/>

<util:list id="patterns">
    <value>org.foo:type=Foo,name=*</value>
    <value>org.foo:type=Bar,name=*</value>
</util:list>
```

The names of the located MBean(s) are logged when DEBUG level logging is enabled.

13.2.2. Notification-publishing Channel Adapter

The notification-publishing channel adapter is relatively simple. It requires only a JMX object name in its configuration, as the following example shows:

```
<context:mbean-export/>

<int-jmx:notification-publishing-channel-adapter id="adapter"
    channel="channel"
    object-name="example.domain:name=publisher"/>
```

It also requires that an `MBeanExporter` be present in the context. That is why the `<context:mbean-export/>` element is also shown in the preceding example.

When messages are sent to the channel for this adapter, the notification is created from the message content. If the payload is a `String`, it is passed as the `message` text for the notification. Any other payload type is passed as the `userData` of the notification.

JMX notifications also have a `type`, and it should be a dot-delimited `String`. There are two ways to provide the `type`. Precedence is always given to a message header value associated with the `JmxHeaders.NOTIFICATION_TYPE` key. Alternatively, you can provide a fallback `default-notification-type` attribute in the configuration, as the following example shows:

```
<context:mbean-export/>

<int-jmx:notification-publishing-channel-adapter id="adapter"
    channel="channel"
    object-name="example.domain:name=publisher"
    default-notification-type="some.default.type"/>
```

13.2.3. Attribute-polling Channel Adapter

The attribute-polling channel adapter is useful when you need to periodically check on some value that is available through an MBean as a managed attribute. You can configured the poller in the same way as any other polling adapter in Spring Integration (or you can rely on the default poller). The `object-name` and the `attribute-name` are required. An MBeanServer reference is also required. However, by default, it automatically checks for a bean named `mbeanServer`, same as the notification-listening channel adapter described earlier. The following example shows how to configure an attribute-polling channel adapter with XML:

```
<int-jmx:attribute-polling-channel-adapter id="adapter"
    channel="channel"
    object-name="example.domain:name=someService"
    attribute-name="InvocationCount">
        <int:poller max-messages-per-poll="1" fixed-rate="5000"/>
</int-jmx:attribute-polling-channel-adapter>
```

13.2.4. Tree-polling Channel Adapter

The tree-polling channel adapter queries the JMX MBean tree and sends a message with a payload that is the graph of objects that matches the query. By default, the MBeans are mapped to primitives and simple objects, such as `Map`, `List`, and arrays. Doing so permits simple transformation to (for example) JSON. An MBeanServer reference is also required. However, by default, it automatically checks for a bean named `mbeanServer`, same as the notification-listening channel adapter described earlier. The following example shows how to configure an tree-polling channel adapter with XML:

```
<int-jmx:tree-polling-channel-adapter id="adapter"
    channel="channel"
    query-name="example.domain:type=*">
        <int:poller max-messages-per-poll="1" fixed-rate="5000"/>
</int-jmx:tree-polling-channel-adapter>
```

The preceding example includes all of the attributes on the selected MBeans. You can filter the attributes by providing an `MBeanObjectConverter` that has an appropriate filter configured. You can

provide the converter as a reference to a bean definition by using the `converter` attribute, or you can use an inner `<bean/>` definition. Spring Integration provides a `DefaultMBeanObjectConverter` that can take a `MBeanAttributeFilter` in its constructor argument.

Spring Integration provides two standard filters. The `NamedFieldsMBeanAttributeFilter` lets you specify a list of attributes to include. The `NotNamedFieldsMBeanAttributeFilter` lets you specify a list of attributes to exclude. You can also implement your own filter.

13.2.5. Operation-invoking Channel Adapter

The operation-invoking channel adapter enables message-driven invocation of any managed operation exposed by an MBean. Each invocation requires the operation name to be invoked and the object name of the target MBean. Both of these must be explicitly provided by adapter configuration or via `JmxHeaders.OBJECT_NAME` and `JmxHeaders.OPERATION_NAME` message headers, respectively:

```
<int-jmx:operation-invoking-channel-adapter id="adapter"
    object-name="example.domain:name=TestBean"
    operation-name="ping"/>
```

Then the adapter only needs to be able to discover the `mbeanServer` bean. If a different bean name is required, then provide the `mbean-server` attribute with a reference.

The payload of the message is mapped to the parameters of the operation, if any. A `Map`-typed payload with `String` keys is treated as name/value pairs, whereas a `List` or array is passed as a simple argument list (with no explicit parameter names). If the operation requires a single parameter value, the payload can represent that single value. Also, if the operation requires no parameters, the payload would be ignored.

If you want to expose a channel for a single common operation to be invoked by messages that need not contain headers, that last option works well.

13.2.6. Operation-invoking Outbound Gateway

Similarly to the operation-invoking channel adapter, Spring Integration also provides an operation-invoking outbound gateway, which you can use when dealing with non-void operations when a return value is required. The return value is sent as the message payload to the `reply-channel` specified by the gateway. The following example shows how to configure an operation-invoking outbound gateway with XML:

```
<int-jmx:operation-invoking-outbound-gateway request-channel="requestChannel"
   reply-channel="replyChannel"
   object-name="o.s.i.jmx.config:type=TestBean,name=testBeanGateway"
   operation-name="testWithReturn"/>
```

If you do not provide the `reply-channel` attribute, the reply message is sent to the channel identified by the `IntegrationMessageHeaderAccessor.REPLY_CHANNEL` header. That header is typically auto-created by the entry point into a message flow, such as any gateway component. However, if the message flow was started by manually creating a Spring Integration message and sending it directly to a channel, you must specify the message header explicitly or use the `reply-channel` attribute.

13.2.7. MBean Exporter

Spring Integration components may themselves be exposed as MBeans when the `IntegrationMBeanExporter` is configured. To create an instance of the `IntegrationMBeanExporter`, define a bean and provide a reference to an `MBeanServer` and a domain name (if desired). You can leave out the domain, in which case the default domain is `org.springframework.integration`. The following example shows how to declare an instance of an `IntegrationMBeanExporter` and an associated `MBeanServer` instance:

```
<int-jmx:mbean-export id="integrationMBeanExporter"
            default-domain="my.company.domain" server="mbeanServer"/>

<bean id="mbeanServer" class=
"org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true"/>
</bean>
```

The MBean exporter is orthogonal to the one provided in Spring core. It registers message channels and message handlers but does not register itself. You can expose the exporter itself (and certain other components in Spring Integration) by using the standard `<context:mbean-export/>` tag. The exporter has some metrics attached to it — for instance, a count of the number of active handlers and the number of queued messages.

It also has a useful operation, as discussed in Orderly Shutdown Managed Operation.

Spring Integration 4.0 introduced the `@EnableIntegrationMBeanExport` annotation to allow for convenient configuration of a default `integrationMbeanExporter` bean of type `IntegrationMBeanExporter` with several useful options at the `@Configuration` class level. The following example shows how to configure this bean:

```
@Configuration
@EnableIntegration
@EnableIntegrationMBeanExport(server = "mbeanServer", managedComponents = "input")
public class ContextConfiguration {

    @Bean
    public MBeanServerFactoryBean mbeanServer() {
        return new MBeanServerFactoryBean();
    }
}
```

If you need to provide more options or have several `IntegrationMBeanExporter` beans (such as for different MBean Servers or to avoid conflicts with the standard Spring `MBeanExporter` — such as through `@EnableMBeanExport`), you can configure an `IntegrationMBeanExporter` as a generic bean.

MBean Object Names

All the `MessageChannel`, `MessageHandler`, and `MessageSource` instances in the application are wrapped by the MBean exporter to provide management and monitoring features. The generated JMX object names for each component type are listed in the following table:

Table 8. MBean Object Names

Component Type	Object Name
MessageChannel	`o.s.i:type=MessageChannel,name=<channelName>`
MessageSource	`o.s.i:type=MessageSource,name=<channelName>,bean=<source>`
MessageHandler	`o.s.i:type=MessageSource,name=<channelName>,bean=<source>`

The `bean` attribute in the object names for sources and handlers takes one of the values in the following table:

Table 9. bean ObjectName Part

Bean Value	Description
endpoint	The bean name of the enclosing endpoint (for example `<service-activator>`), if there is one
anonymous	An indication that the enclosing endpoint did not have a user-specified bean name, so the JMX name is the input channel name.
internal	For well known Spring Integration default components
handler/source	None of the above. Fall back to the `toString()` method of the object being monitored (handler or source)

You can append custom elements to the object name by providing a reference to a `Properties` object in the `object-name-static-properties` attribute.

Also, since Spring Integration 3.0, you can use a custom `ObjectNamingStrategy` by setting the `object-naming-strategy` attribute. Doing so permits greater control over the naming of the MBeans, such as grouping all integration MBeans under an 'Integration' type. The following example shows one possible custom naming strategy implementation:

```
public class Namer implements ObjectNamingStrategy {

    private final ObjectNamingStrategy realNamer = new KeyNamingStrategy();
    @Override
    public ObjectName getObjectName(Object managedBean, String beanKey) throws
MalformedObjectNameException {
        String actualBeanKey = beanKey.replace("type=",
"type=Integration,componentType=");
        return realNamer.getObjectName(managedBean, actualBeanKey);
    }

}
```

The `beanKey` argument is a `String` that contain the standard object name, beginning with the `default-domain` and including any additional static properties. The preceding example moves the standard `type` part to `componentType` and sets the `type` to 'Integration', enabling selection of all Integration MBeans in one query:`my.domain:type=Integration,*`. Doing so also groups the beans under one tree entry under the domain in such tools as VisualVM.

The default naming strategy is a `MetadataNamingStrategy`. The exporter propagates the `default-domain` to that object to let it generate a fallback object name if parsing of the bean key fails. If your custom naming strategy is a `MetadataNamingStrategy` (or a subclass of it), the exporter does not propagate the `default-domain`. You must configure it on your strategy bean.

Starting with version 5.1; any bean names (represented by the `name` key in the object name) will be quoted if they contain any characters that are not allowed in a Java identifier (or period `.`).

JMX Improvements

Version 4.2 introduced some important improvements, representing a fairly major overhaul to the JMX support in the framework. These resulted in a significant performance improvement of the JMX statistics collection and much more control thereof. However, it has some implications for user code in a few specific (uncommon) situations. These changes are detailed below, with a caution where necessary.

Metrics Capture

Previously, `MessageSource`, `MessageChannel`, and `MessageHandler` metrics were captured by wrapping the object in a JDK dynamic proxy to intercept appropriate method calls and capture the statistics. The proxy was added when an integration MBean exporter was declared in the context.

Now, the statistics are captured by the beans themselves. See Metrics and Management for more information.

This change means that you no longer automatically get an MBean or statistics for custom `MessageHandler` implementations, unless those custom handlers extend `AbstractMessageHandler`. The simplest way to resolve this is to extend `AbstractMessageHandler`. If you cannot do so, another work around is to implement the `MessageHandlerMetrics` interface. For convenience, a `DefaultMessageHandlerMetrics` is provided to capture and report statistics. You should invoke the `beforeHandle` and `afterHandle` at the appropriate times. Your `MessageHandlerMetrics` methods can then delegate to this object to obtain each statistic. Similarly, `MessageSource` implementations must extend `AbstractMessageSource` or implement `MessageSourceMetrics`. Message sources capture only a count, so there is no provided convenience class. You should maintain the count in an `AtomicLong` field.

The removal of the proxy has two additional benefits:

- Stack traces in exceptions are reduced (when JMX is enabled) because the proxy is not on the stack
- Cases where two MBeans were exported for the same bean now only export a single MBean with consolidated attributes and operations (see the MBean consolidation bullet, later).

Resolution

`System.nanoTime()` (rather than `System.currentTimeMillis()`) is now used to capture times . This may provide more accuracy on some JVMs, espcially when you expect durations of less than one millisecond.

Setting Initial Statistics Collection State

Previously, when JMX was enabled, all sources, channels, and handlers captured statistics. You can now control whether the statistics are enabled on an individual component. Further, you can capture simple counts on `MessageChannel` instances and `MessageHandler` instances instead of capturing the complete time-based statistics. This can have significant performance implications, because you can selectively configure where you need detailed statistics and enable and disable collection at runtime.

See Metrics and Management.

@IntegrationManagedResource

Similar to the `@ManagedResource` annotation, the `@IntegrationManagedResource` marks a class as being eligible to be exported as an MBean. However, it is exported only if the application context has an `IntegrationMBeanExporter`.

Certain Spring Integration classes (in the `org.springframework.integration`) package) that were previously annotated with `@ManagedResource` are now annotated with both `@ManagedResource` and `@IntegrationManagedResource`. This is for backwards compatibility (see the next item). Such MBeans are exported by any context `MBeanServer` or by an `IntegrationMBeanExporter` (but not both — if both exporters are present, the bean is exported by the integration exporter if the bean

matches a `managed-components` pattern).

Consolidated MBeans

Certain classes within the framework (mapping routers, for example) have additional attributes and operations over and above those provided by metrics and `Lifecycle`. We use a `Router` as an example here.

Previously, beans of these types were exported as two distinct MBeans:

- The metrics MBean (with an object name such as `intDomain:type=MessageHandler,name=myRouter,bean=endpoint`). This MBean had metrics attributes and metrics/Lifecycle operations.
- A second MBean (with an object name such as `ctxDomain:name=org.springframework.integration.config.RouterFactoryBean#0`,type=MethodInvokingRouter`) was exported with the channel mappings attribute and operations.

  Now the attributes and operations are consolidated into a single MBean. The object name depends on the exporter. If exported by the integration MBean exporter, the object name is, for example: `intDomain:type=MessageHandler,name=myRouter,bean=endpoint`. If exported by another exporter, the object name is, for example: `ctxDomain:name=org.springframework.integration.config.RouterFactoryBean#0,type=MethodInvokingRouter`. There is no difference between these MBeans (aside from the object name), except that the statistics are not enabled (the attributes are `0`) by exporters other than the integration exporter. You can enable statistics at runtime by using the JMX operations. When exported by the integration MBean exporter, the initial state can be managed as described earlier.

If you currently use the second MBean to change, for example, channel mappings and you use the integration MBean exporter, note that the object name has changed because of the MBean consolidation. There is no change if you are not using the integration MBean exporter.

MBean Exporter Bean Name Patterns

Previously, the `managed-components` patterns were inclusive only. If a bean name matched one of the patterns, it would be included. Now, the pattern can be negated by prefixing it with `!`. For example, `!thing*, things` matches all bean names that do not start with `thing` except `things`. Patterns are evaluated left to right. The first match (positive or negative) wins, and then no further patterns are applied.

The addition of this syntax to the pattern causes one possible (although perhaps unlikely) problem. If you have a bean named `"!thing"` and you included a pattern of `!thing` in your MBean exporter's `managed-components` patterns, it no longer matches; the pattern now matches all beans not named `thing`. In this case, you can escape the `!` in the pattern with `\`. The `\!thing` pattern matches a bean named `!thing`.

IntegrationMBeanExporter changes

The `IntegrationMBeanExporter` no longer implements `SmartLifecycle`. This means that `start()` and `stop()` operations are no longer available to register and unregister MBeans. The MBeans are now registered during context initialization and unregistered when the context is destroyed.

Orderly Shutdown Managed Operation

The MBean exporter provides a JMX operation to shut down the application in an orderly manner, intended for use before terminating the JVM. The following example shows how to use it:

```
public void stopActiveComponents(long howLong)
```

Its use and operation are described in Orderly Shutdown.

13.3. Message History

The key benefit of a messaging architecture is loose coupling such that participating components do not maintain any awareness about one another. This fact alone makes an application extremely flexible, letting you change components without affecting the rest of the flow, change messaging routes, change message consuming styles (polling versus event driven), and so on. However, this unassuming style of architecture could prove to be difficult when things go wrong. When debugging, you probably want as much information (its origin, the channels it has traversed, and other details) about the message as you can get.

Message history is one of those patterns that helps by giving you an option to maintain some level of awareness of a message path either for debugging purposes or for maintaining an audit trail. Spring integration provides a simple way to configure your message flows to maintain the message history by adding a header to the message and updating that header every time a message passes through a tracked component.

13.3.1. Message History Configuration

To enable message history, you need only define the `message-history` element in your configuration, as shown in the following example:

```
<int:message-history/>
```

Now every named component (component that has an 'id' defined) is tracked. The framework sets the 'history' header in your message. Its value a `List<Properties>`.

Consider the following configuration example:

```
<int:gateway id="sampleGateway"
    service-interface=
"org.springframework.integration.history.sample.SampleGateway"
    default-request-channel="bridgeInChannel"/>

<int:chain id="sampleChain" input-channel="chainChannel" output-channel=
"filterChannel">
  <int:header-enricher>
    <int:header name="baz" value="baz"/>
  </int:header-enricher>
</int:chain>
```

The preceding configuration produces a simple message history structure, with output similar to the following:

```
[{name=sampleGateway, type=gateway, timestamp=1283281668091},
 {name=sampleChain, type=chain, timestamp=1283281668094}]
```

To get access to message history, you need only access the `MessageHistory` header. The folloiwng example shows how to do so:

```
Iterator<Properties> historyIterator =
    message.getHeaders().get(MessageHistory.HEADER_NAME, MessageHistory.class)
.iterator();
assertTrue(historyIterator.hasNext());
Properties gatewayHistory = historyIterator.next();
assertEquals("sampleGateway", gatewayHistory.get("name"));
assertTrue(historyIterator.hasNext());
Properties chainHistory = historyIterator.next();
assertEquals("sampleChain", chainHistory.get("name"));
```

You might not want to track all of the components. To limit the history to certain components based on their names, you can provide the `tracked-components` attribute and specify a comma-delimited list of component names and patterns that match the components you want to track. The following example shows how to do so:

```
<int:message-history tracked-components="*Gateway, sample*, aName"/>
```

In the preceding example, message history is maintained only for the components that end with 'Gateway', start with 'sample', or match the name, 'aName', exactly.

Starting with version 4.0, you can also use the `@EnableMessageHistory` annotation in a `@Configuration` class. In addition, the `MessageHistoryConfigurer` bean is now exposed as a JMX MBean by the `IntegrationMBeanExporter` (see MBean Exporter), letting you change the patterns at runtime. Note, however, that the bean must be stopped (turning off message history) in order to change the patterns. This feature might be useful to temporarily turn on history to analyze a system. The MBean's object name is `<domain>:name=messageHistoryConfigurer,type=MessageHistoryConfigurer`.

If multiple beans (declared by `@EnableMessageHistory` and `<message-history/>`) exist, they must all have identical component name patterns (when trimmed and sorted). Do not use a generic `<bean/>` definition for the `MessageHistoryConfigurer`.

By definition, the message history header is immutable (you cannot re-write history). Therefore, when writing message history values, the components either create new messages (when the component is an origin) or they copy the history from a request message, modifying it and setting the new list on a reply message. In either case, the values can be appended even if the message itself is crossing thread boundaries. That means that the history values can greatly simplify debugging in an asynchronous message flow.

13.4. Message Store

The *Enterprise Integration Patterns* (EIP) book identifies several patterns that have the ability to buffer messages. For example, an aggregator buffers messages until they can be released, and a `QueueChannel` buffers messages until consumers explicitly receive those messages from that channel. Because of the failures that can occur at any point within your message flow, EIP components that buffer messages also introduce a point where messages could be lost.

To mitigate the risk of losing messages, EIP defines the message store pattern, which lets EIP components store messages, typically in some type of persistent store (such as an RDBMS).

Spring Integration provides support for the message store pattern by:

- Defining an `org.springframework.integration.store.MessageStore` strategy interface
- Providing several implementations of this interface
- Exposing a `message-store` attribute on all components that have the capability to buffer messages so that you can inject any instance that implements the `MessageStore` interface.

Details on how to configure a specific message store implementation and how to inject a `MessageStore` implementation into a specific buffering component are described throughout the manual (see the specific component, such as QueueChannel, Aggregator, Delayer, and others). The following pair of examples show how to add a reference to a message store for a `QueueChannel` and for an aggregator:

Example 4. QueueChannel

```
<int:channel id="myQueueChannel">
    <int:queue message-store="refToMessageStore"/>
<int:channel>
```

Example 5. Aggregator

```
<int:aggregator ... message-store="refToMessageStore"/>
```

By default, messages are stored in-memory by using `o.s.i.store.SimpleMessageStore`, an implementation of `MessageStore`. That might be fine for development or simple low-volume environments where the potential loss of non-persistent messages is not a concern. However, the typical production application needs a more robust option, not only to mitigate the risk of message loss but also to avoid potential out-of-memory errors. Therefore, we also provide `MessageStore` implementations for a variety of data-stores. The following is a complete list of supported implementations:

- JDBC Message Store: Uses an RDBMS to store messages
- Redis Message Store: Uses a Redis key/value datastore to store messages
- MongoDB Message Store: Uses a MongoDB document store to store messages
- Gemfire Message Store: Uses a Gemfire distributed cache to store messages

However, be aware of some limitations while using persistent implementations of the `MessageStore`.

The Message data (payload and headers) is serialized and deserialized by using different serialization strategies, depending on the implementation of the `MessageStore`. For example, when using `JdbcMessageStore`, only `Serializable` data is persisted by default. In this case, non-Serializable headers are removed before serialization occurs. Also, be aware of the protocol-specific headers that are injected by transport adapters (such as FTP, HTTP, JMS, and others). For example, `<http:inbound-channel-adapter/>` maps HTTP headers into message headers, and one of them is an `ArrayList` of non-serializable `org.springframework.http.MediaType` instances. However, you can inject your own implementation of the `Serializer` and `Deserializer` strategy interfaces into some `MessageStore` implementations (such as `JdbcMessageStore`) to change the behavior of serialization and deserialization.

Pay special attention to the headers that represent certain types of data. For example, if one of the headers contains an instance of some Spring bean, upon deserialization, you may end up with a different instance of that bean, which directly affects some of the implicit headers created by the framework (such as `REPLY_CHANNEL` or `ERROR_CHANNEL`). Currently, they are not serializable, but, even if they were, the deserialized channel would not represent the expected instance.

Beginning with Spring Integration version 3.0, you can resolve this issue with a header enricher configured to replace these headers with a name after registering the channel with the `HeaderChannelRegistry`.

Also, consider what happens when you configure a message-flow as follows: gateway → queue-channel (backed by a persistent Message Store) → service-activator. That gateway creates a temporary reply channel, which is lost by the time the service-activator's poller reads from the queue. Again, you can use the header enricher to replace the headers with a `String` representation.

For more information, see Header Enricher.

Spring Integration 4.0 introduced two new interfaces:

- `ChannelMessageStore`: To implement operations specific for `QueueChannel` instances
- `PriorityCapableChannelMessageStore`: To mark `MessageStore` implementations to be used for `PriorityChannel` instances and to provide priority order for persisted messages.

The real behavior depends on the implementation. The framework provides the following implementations, which can be used as a persistent `MessageStore` for `QueueChannel` and `PriorityChannel`:

- Redis Channel Message Stores
- MongoDB Channel Message Store
- Backing Message Channels

Caution about `SimpleMessageStore`

Starting with version 4.1, the `SimpleMessageStore` no longer copies the message group when calling `getMessageGroup()`. For large message groups, this was a significant performance problem. 4.0.1 introduced a boolean `copyOnGet` property that lets you control this behavior. When used internally by the aggregator, this property was set to `false` to improve performance. It is now `false` by default.

Users accessing the group store outside of components such as aggregators now get a direct reference to the group being used by the aggregator instead of a copy. Manipulation of the group outside of the aggregator may cause unpredictable results.

For this reason, you should either not perform such manipulation or set the `copyOnGet` property to `true`.

13.4.1. Using `MessageGroupFactory`

Starting with version 4.3, some `MessageGroupStore` implementations can be injected with a custom `MessageGroupFactory` strategy to create and customize the `MessageGroup` instances used by the `MessageGroupStore`. This defaults to a `SimpleMessageGroupFactory`, which produces `SimpleMessageGroup` instances based on the `GroupType.HASH_SET` (`LinkedHashSet`) internal collection. Other possible options are `SYNCHRONISED_SET` and `BLOCKING_QUEUE`, where the last one can be used to reinstate the previous `SimpleMessageGroup` behavior. Also the `PERSISTENT` option is available. See the next section for more information. Starting with version 5.0.1, the `LIST` option is also available for when the order and uniqueness of messages in the group does not matter.

13.4.2. Persistent `MessageGroupStore` and Lazy-load

Starting with version 4.3, all persistent `MessageGroupStore` instances retrieve `MessageGroup` instances and their `messages` from the store in the lazy-load manner. In most cases, it is useful for the correlation `MessageHandler` instances (see Aggregator and Resequencer), when it would add overhead to load entire the `MessageGroup` from the store on each correlation operation.

You can use the `AbstractMessageGroupStore.setLazyLoadMessageGroups(false)` option to switch off the lazy-load behavior from the configuration.

Our performance tests for lazy-load on MongoDB `MessageStore` (MongoDB Message Store) and `<aggregator>` (Aggregator) use a custom `release-strategy` similar to the following:

```
<int:aggregator input-channel="inputChannel"
                output-channel="outputChannel"
                message-store="mongoStore"
                release-strategy-expression="size() == 1000"/>
```

It produces results similar to the following for 1000 simple messages:

```
...
StopWatch 'Lazy-Load Performance': running time (millis) = 38918
-----------------------------------------
ms     %     Task name
-----------------------------------------
02652  007%  Lazy-Load
36266  093%  Eager
...
```

13.5. Metadata Store

Many external systems, services, or resources are not transactional (Twitter, RSS, file systems, and so on), and there is no any ability to mark the data as read. Also, sometimes, you may need to implement the Enterprise Integration Pattern idempotent receiver in some integration solutions. To achieve this goal and store some previous state of the endpoint before the next interaction with external system or to deal with the next message, Spring Integration provides the metadata store component as an an implementation of the `org.springframework.integration.metadata.MetadataStore` interface with a general key-value contract.

The metadata store is designed to store various types of generic metadata (for example, the published date of the last feed entry that has been processed) to help components such as the feed adapter deal with duplicates. If a component is not directly provided with a reference to a `MetadataStore`, the algorithm for locating a metadata store is as follows: First, look for a bean with a `metadataStore` ID in the application context. If one is found, use it. Otherwise, create a new instance of `SimpleMetadataStore`, which is an in-memory implementation that persists only metadata within the lifecycle of the currently running application context. This means that, upon restart, you may end up with duplicate entries.

If you need to persist metadata between application context restarts, the framework provides the following persistent `MetadataStores`:

- `PropertiesPersistingMetadataStore`
- Gemfire Metadata Store
- JDBC Metadata Store
- MongoDB Metadata Store
- Redis Metadata Store
- Zookeeper Metadata Store

The `PropertiesPersistingMetadataStore` is backed by a properties file and a `PropertiesPersister`.

By default, it persists only the state when the application context is closed normally. It implements `Flushable` so that you can persist the state at will, by invoking `flush()`. The following example shows how to configure a 'PropertiesPersistingMetadataStore' with XML:

```
<bean id="metadataStore"
    class="org.springframework.integration.metadata.PropertiesPersistingMetadataStore
"/>
```

Alternatively, you can provide your own implementation of the `MetadataStore` interface (for example, `JdbcMetadataStore`) and configure it as a bean in the application context.

Starting with version 4.0, `SimpleMetadataStore`, `PropertiesPersistingMetadataStore`, and `RedisMetadataStore` implement `ConcurrentMetadataStore`. These provide for atomic updates and can be used across multiple component or application instances.

13.5.1. Idempotent Receiver and Metadata Store

The metadata store is useful for implementing the EIP idempotent receiver pattern when there is need to filter an incoming message if it has already been processed and you can discard it or perform some other logic on discarding. The following configuration shows an example of how to do so:

```
<int:filter input-channel="serviceChannel"
            output-channel="idempotentServiceChannel"
            discard-channel="discardChannel"
            expression="@metadataStore.get(headers.businessKey) == null"/>

<int:publish-subscribe-channel id="idempotentServiceChannel"/>

<int:outbound-channel-adapter channel="idempotentServiceChannel"
                              expression="@metadataStore.put(headers.businessKey,
'')"/>

<int:service-activator input-channel="idempotentServiceChannel" ref="service"/>
```

The `value` of the idempotent entry may be an expiration date, after which that entry should be removed from metadata store by some scheduled reaper.

See also Idempotent Receiver Enterprise Integration Pattern.

13.5.2. `MetadataStoreListener`

Some metadata stores (currently only zookeeper) support registering a listener to receive events when items change, as the following example shows:

```
public interface MetadataStoreListener {

    void onAdd(String key, String value);

    void onRemove(String key, String oldValue);

    void onUpdate(String key, String newValue);
}
```

See the Javadoc for more information. The `MetadataStoreListenerAdapter` can be subclassed if you are interested only in a subset of events.

13.6. Control Bus

As described in the *Enterprise Integration Patterns* (EIP) book, the idea behind the control bus is that the same messaging system can be used for monitoring and managing the components within the framework as is used for "application-level" messaging. In Spring Integration, we build upon the adapters described above so that you can send messages as a means of invoking exposed operations.

The following example shows how to configure a control bus with XML:

```
<int:control-bus input-channel="operationChannel"/>
```

The control bus has an input channel that can be accessed for invoking operations on the beans in the application context. It also has all the common properties of a service activating endpoint. For example, you can specify an output channel if the result of the operation has a return value that you want to send on to a downstream channel.

The control bus runs messages on the input channel as Spring Expression Language (SpEL) expressions. It takes a message, compiles the body to an expression, adds some context, and then runs it. The default context supports any method that has been annotated with `@ManagedAttribute` or `@ManagedOperation`. It also supports the methods on Spring's `Lifecycle` interface (and its `Pausable` extension since version 5.2), and it supports methods that are used to configure several of Spring's `TaskExecutor` and `TaskScheduler` implementations. The simplest way to ensure that your own methods are available to the control bus is to use the `@ManagedAttribute` or `@ManagedOperation` annotations. Since those annotations are also used for exposing methods to a JMX MBean registry, they offer a convenient by-product: Often, the same types of operations you want to expose to the control bus are reasonable for exposing through JMX). Resolution of any particular instance within the application context is achieved in the typical SpEL syntax. To do so, provide the bean name with the SpEL prefix for beans (`@`). For example, to execute a method on a Spring Bean, a client could send a message to the operation channel as follows:

```
Message operation = MessageBuilder.withPayload("@myServiceBean.shutdown()").build
();
operationChannel.send(operation)
```

The root of the context for the expression is the `Message` itself, so you also have access to the `payload` and `headers` as variables within your expression. This is consistent with all the other expression support in Spring Integration endpoints.

With Java annotations, you can configured the control bus as follows:

```
@Bean
@ServiceActivator(inputChannel = "operationChannel")
public ExpressionControlBusFactoryBean controlBus() {
    return new ExpressionControlBusFactoryBean();
}
```

Similarly, you can configure Java DSL flow definitions as follows:

```
@Bean
public IntegrationFlow controlBusFlow() {
    return IntegrationFlows.from("controlBus")
              .controlBus()
              .get();
}
```

If you prefer to use lambdas with automatic `DirectChannel` creation, you can create a control bus as follows:

```
@Bean
public IntegrationFlow controlBus() {
    return IntegrationFlowDefinition::controlBus;
}
```

In this case, the channel is named `controlBus.input`.

13.7. Orderly Shutdown

As described in "MBean Exporter", the MBean exporter provides a JMX operation called `stopActiveComponents`, which is used to stop the application in an orderly manner. The operation has

a single `Long` parameter. The parameter indicates how long (in milliseconds) the operation waits to allow in-flight messages to complete. The operation works as follows:

1. Call `beforeShutdown()` on all beans that implement `OrderlyShutdownCapable`.

   Doing so lets such components prepare for shutdown. Examples of components that implement this interface and what they do with this call include JMS and AMQP message-driven adapters that stop their listener containers, TCP server connection factories that stop accepting new connections (while keeping existing connections open), TCP inbound endpoints that drop (log) any new messages received, and HTTP inbound endpoints that return `503 - Service Unavailable` for any new requests.

2. Stop any active channels, such as JMS- or AMQP-backed channels.
3. Stop all `MessageSource` instances.
4. Stop all inbound `MessageProducer` s (that are not `OrderlyShutdownCapable`).
5. Wait for any remaining time left, as defined by the value of the `Long` parameter passed in to the operation.

   Doing so lets any in-flight messages complete their journeys. It is therefore important to select an appropriate timeout when invoking this operation.

6. Call `afterShutdown()` on all `OrderlyShutdownCapable` components.

   Doing so lets such components perform final shutdown tasks (closing all open sockets, for example).

As discussed in Orderly Shutdown Managed Operation, this operation can be invoked by using JMX. If you wish to programmatically invoke the method, you need to inject or otherwise get a reference to the `IntegrationMBeanExporter`. If no `id` attribute is provided on the `<int-jmx:mbean-export/>` definition, the bean has a generated name. This name contains a random component to avoid `ObjectName` collisions if multiple Spring Integration contexts exist in the same JVM (`MBeanServer`).

For this reason, if you wish to invoke the method programmatically, we recommend that you provide the exporter with an `id` attribute so that you can easily access it in the application context.

Finally, the operation can be invoked by using the `<control-bus>` element. See the monitoring Spring Integration sample application for details.

The algorithm described earlier was improved in version 4.1. Previously, all task executors and schedulers were stopped. This could cause mid-flow messages in `QueueChannel` instances to remain. Now the shutdown leaves pollers running, to let these messages be drained and processed.

13.8. Integration Graph

Starting with version 4.3, Spring Integration provides access to an application's runtime object model, which can, optionally, include component metrics. It is exposed as a graph, which may be used to visualize the current state of the integration application. The

`o.s.i.support.management.graph` package contains all the required classes to collect, build, and render the runtime state of Spring Integration components as a single tree-like `Graph` object. The `IntegrationGraphServer` should be declared as a bean to build, retrieve, and refresh the `Graph` object. The resulting `Graph` object can be serialized to any format, although JSON is flexible and convenient to parse and represent on the client side. A Spring Integration application with only the default components would expose a graph as follows:

```
{
  "contentDescriptor" : {
    "providerVersion" : "5.3.0.RC1",
    "providerFormatVersion" : 1.2,
    "provider" : "spring-integration",
    "name" : "myAppName:1.0"
  },
  "nodes" : [ {
    "nodeId" : 1,
    "componentType" : "null-channel",
    "integrationPatternType" : "null_channel",
    "integrationPatternCategory" : "messaging_channel",
    "properties" : { },
    "sendTimers" : {
      "successes" : {
        "count" : 1,
        "mean" : 0.0,
        "max" : 0.0
      },
      "failures" : {
        "count" : 0,
        "mean" : 0.0,
        "max" : 0.0
      }
    },
    "receiveCounters" : {
      "successes" : 0,
      "failures" : 0
    },
    "name" : "nullChannel"
  }, {
    "nodeId" : 2,
    "componentType" : "publish-subscribe-channel",
    "integrationPatternType" : "publish_subscribe_channel",
    "integrationPatternCategory" : "messaging_channel",
    "properties" : { },
    "sendTimers" : {
      "successes" : {
        "count" : 1,
        "mean" : 7.807002,
        "max" : 7.807002
      },
      "failures" : {
        "count" : 0,
        "mean" : 0.0,
        "max" : 0.0
      }
    },
    "name" : "errorChannel"
```

```
  }, {
    "nodeId" : 3,
    "componentType" : "logging-channel-adapter",
    "integrationPatternType" : "outbound_channel_adapter",
    "integrationPatternCategory" : "messaging_endpoint",
    "properties" : { },
    "output" : null,
    "input" : "errorChannel",
    "sendTimers" : {
      "successes" : {
        "count" : 1,
        "mean" : 6.742722,
        "max" : 6.742722
      },
      "failures" : {
        "count" : 0,
        "mean" : 0.0,
        "max" : 0.0
      }
    },
    "name" : "errorLogger"
  } ],
  "links" : [ {
    "from" : 2,
    "to" : 3,
    "type" : "input"
  } ]
}
```

Version 5.2 has deprecated the legacy metrics in favor of Micrometer meters as discussed Metrics Management. While not shown above, the legacy metrics (under the `stats` child node) will continue to appear in the graph, but with an extra child node `"deprecated" : "stats are deprecated in favor of sendTimers and receiveCounters"`.

With some JSON serializers, you can suppress the inclusion of legacy statistics using several techniques; for example, with Jackson, you can register a `SimpleModule` configured with a `NullSerializer` with the `ObjectMapper`:

```
objectMapper.registerModule(new SimpleModule()
        .addSerializer(IntegrationNode.Stats.class, NullSerializer.instance));
```

The resulting json contains `"stats" : null`.

In the preceding example, the graph consists of three top-level elements.

The `contentDescriptor` graph element contains general information about the application providing the data. The `name` can be customized on the `IntegrationGraphServer` bean or in the `spring.application.name` application context environment property. Other properties are provided by the framework and let you distinguish a similar model from other sources.

The `links` graph element represents connections between nodes from the `nodes` graph element and, therefore, between integration components in the source Spring Integration application. For example, from a `MessageChannel` to an `EventDrivenConsumer` with some `MessageHandler` or from an `AbstractReplyProducingMessageHandler` to a `MessageChannel`. For convenience and to let you determine a link's purpose, the model includes the `type` attribute. The possible types are:

- `input`: Identifies the direction from `MessageChannel` to the endpoint, `inputChannel`, or `requestChannel` property
- `output`: The direction from the `MessageHandler`, `MessageProducer`, or `SourcePollingChannelAdapter` to the `MessageChannel` through an `outputChannel` or `replyChannel` property
- `error`: From `MessageHandler` on `PollingConsumer` or `MessageProducer` or `SourcePollingChannelAdapter` to the `MessageChannel` through an `errorChannel` property;
- `discard`: From `DiscardingMessageHandler` (such as `MessageFilter`) to the `MessageChannel` through an `errorChannel` property.
- `route`: From `AbstractMappingMessageRouter` (such as `HeaderValueRouter`) to the `MessageChannel`. Similar to `output` but determined at run-time. May be a configured channel mapping or a dynamically resolved channel. Routers typically retain only up to 100 dynamic routes for this purpose, but you can modify this value by setting the `dynamicChannelLimit` property.

The information from this element can be used by a visualization tool to render connections between nodes from the `nodes` graph element, where the `from` and `to` numbers represent the value from the `nodeId` property of the linked nodes. For example, the `link` element can be used to determine the proper `port` on the target node.

The following "text image" shows the relationships between the types:

The nodes graph element is perhaps the most interesting, because its elements contain not only the runtime components with their componentType instances and name values but can also optionally contain metrics exposed by the component. Node elements contain various properties that are generally self-explanatory. For example, expression-based components include the expression property that contains the primary expression string for the component. To enable the metrics, add an @EnableIntegrationManagement to a @Configuration class or add an <int:management/> element to your XML configuration. See Metrics and Management for complete information.

The nodeId represents a unique incremental identifier to let you distinguish one component from another. It is also used in the links element to represent a relationship (connection) of this component to others, if any. The input and output attributes are for the inputChannel and outputChannel properties of the AbstractEndpoint, MessageHandler, SourcePollingChannelAdapter, or MessageProducerSupport. See the next section for more information.

Starting with version 5.1, the IntegrationGraphServer accepts a Function<NamedComponent, Map<String, Object>> additionalPropertiesCallback for population of additional properties on the IntegrationNode for a particular NamedComponent. For example you can expose the SmartLifecycle autoStartup and running properties into the target graph:

```
server.setAdditionalPropertiesCallback(namedComponent -> {
            Map<String, Object> properties = null;
            if (namedComponent instanceof SmartLifecycle) {
                SmartLifecycle smartLifecycle = (SmartLifecycle) namedComponent;
                properties = new HashMap<>();
                properties.put("auto-startup", smartLifecycle.isAutoStartup());
                properties.put("running", smartLifecycle.isRunning());
            }
            return properties;
        });
```

13.8.1. Graph Runtime Model

Spring Integration components have various levels of complexity. For example, any polled MessageSource also has a SourcePollingChannelAdapter and a MessageChannel to which to periodically send messages from the source data. Other components might be middleware request-reply components (such as JmsOutboundGateway) with a consuming AbstractEndpoint to subscribe to (or poll) the requestChannel (input) for messages, and a replyChannel (output) to produce a reply message to send downstream. Meanwhile, any MessageProducerSupport implementation (such as ApplicationEventListeningMessageProducer) wraps some source protocol listening logic and sends messages to the outputChannel.

Within the graph, Spring Integration components are represented by using the IntegrationNode class hierarchy, which you can find in the o.s.i.support.management.graph package. For example, you can use the ErrorCapableDiscardingMessageHandlerNode for the AggregatingMessageHandler (because it has a discardChannel option) and can produce errors when consuming from a PollableChannel by using a PollingConsumer. Another example is CompositeMessageHandlerNode — for

a `MessageHandlerChain` when subscribed to a `SubscribableChannel` by using an `EventDrivenConsumer`.

The `@MessagingGateway` (see Messaging Gateways) provides nodes for each of its method, where the `name` attribute is based on the gateway's bean name and the short method signature. Consider the following example of a gateway:

```
@MessagingGateway(defaultRequestChannel = "four")
public interface Gate {

    void foo(String foo);

    void foo(Integer foo);

    void bar(String bar);

}
```

The preceding gateway produces nodes similar to the following:

```
{
  "nodeId" : 10,
  "name" : "gate.bar(class java.lang.String)",
  "stats" : null,
  "componentType" : "gateway",
  "integrationPatternType" : "gateway",
  "integrationPatternCategory" : "messaging_endpoint",
  "output" : "four",
  "errors" : null
},
{
  "nodeId" : 11,
  "name" : "gate.foo(class java.lang.String)",
  "stats" : null,
  "componentType" : "gateway",
  "integrationPatternType" : "gateway",
  "integrationPatternCategory" : "messaging_endpoint",
  "output" : "four",
  "errors" : null
},
{
  "nodeId" : 12,
  "name" : "gate.foo(class java.lang.Integer)",
  "stats" : null,
  "componentType" : "gateway",
  "integrationPatternType" : "gateway",
  "integrationPatternCategory" : "messaging_endpoint",
  "output" : "four",
  "errors" : null
}
```


You can use this `IntegrationNode` hierarchy for parsing the graph model on the client side as well as to understand the general Spring Integration runtime behavior. See also Programming Tips and Tricks for more information.

Version 5.3 introduced an `IntegrationPattern` abstraction and all out-of-the-box components, which represent an Enterprise Integration Pattern (EIP), implement this abstraction and provide an `IntegrationPatternType` enum value. This information can be useful for some categorizing logic in the target application or, being exposed into the graph node, it can be used by a UI to determine how to draw the component.

13.9. Integration Graph Controller

If your application is web-based (or built on top of Spring Boot with an embedded web container) and the Spring Integration HTTP or WebFlux module (see HTTP Support and WebFlux Support, respectively) is present on the classpath, you can use a `IntegrationGraphController` to expose the `IntegrationGraphServer` functionality as a REST service. For this purpose, the

`@EnableIntegrationGraphController` and `@Configuration` class annotations and the `<int-http:graph-controller/>` XML element are available in the HTTP module. Together with the `@EnableWebMvc` annotation (or `<mvc:annotation-driven/>` for XML definitions), this configuration registers an `IntegrationGraphController` `@RestController` where its `@RequestMapping.path` can be configured on the `@EnableIntegrationGraphController` annotation or `<int-http:graph-controller/>` element. The default path is `/integration`.

The `IntegrationGraphController` `@RestController` provides the following services:

- `@GetMapping(name = "getGraph")`: To retrieve the state of the Spring Integration components since the last `IntegrationGraphServer` refresh. The `o.s.i.support.management.graph.Graph` is returned as a `@ResponseBody` of the REST service.
- `@GetMapping(path = "/refresh", name = "refreshGraph")`: To refresh the current `Graph` for the actual runtime state and return it as a REST response. It is not necessary to refresh the graph for metrics. They are provided in real-time when the graph is retrieved. Refresh can be called if the application context has been modified since the graph was last retrieved. In that case, the graph is completely rebuilt.

You can set security and cross-origin restrictions for the `IntegrationGraphController` with the standard configuration options and components provided by the Spring Security and Spring MVC projects. The following example achieves those goals:

```
<mvc:annotation-driven />

<mvc:cors>
    <mvc:mapping path="/myIntegration/**"
                 allowed-origins="http://localhost:9090"
                 allowed-methods="GET" />
</mvc:cors>

<security:http>
    <security:intercept-url pattern="/myIntegration/**" access="ROLE_ADMIN" />
</security:http>


<int-http:graph-controller path="/myIntegration" />
```

The following example shows how to do the same thing with Java configuration:

```
@Configuration
@EnableWebMvc // or @EnableWebFlux
@EnableWebSecurity // or @EnableWebFluxSecurity
@EnableIntegration
@EnableIntegrationGraphController(path = "/testIntegration", allowedOrigins=
"http://localhost:9090")
public class IntegrationConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
               .antMatchers("/testIntegration/**").hasRole("ADMIN")
            // ...
            .formLogin();
    }

    //...

}
```

Note that, for convenience, the `@EnableIntegrationGraphController` annotation provides an `allowedOrigins` attribute. This provides `GET` access to the `path`. For more sophistication, you can configure the CORS mappings by using standard Spring MVC mechanisms.

Integration Endpoints

This section covers the various channel adapters and messaging gateways provided by Spring Integration to support message-based communication with external systems.

Each system, from AMQP to Zookeeper, has its own integration requirements, and this section covers them.

Chapter 14. Endpoint Quick Reference Table

As discussed in the earlier sections, Spring Integration provides a number of endpoints used to interface with external systems, file systems, and others.

For transparent dependency management Spring Integration provides a bill-of-materials POM to be imported into the Maven configuration:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.integration</groupId>
            <artifactId>spring-integration-bom</artifactId>
            <version>5.3.0.RC1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

To recap:

- Inbound channel adapters are used for one-way integration to bring data into the messaging application.
- Outbound channel adapters are used for one-way integration to send data out of the messaging application.
- Inbound gateways are used for a bidirectional integration flow, where some other system invokes the messaging application and receives a reply.
- Outbound Gateways are used for a bidirectional integration flow, where the messaging application invokes some external service or entity and expects a result.

The following table summarizes the various endpoints with quick links to the appropriate chapter.

Table 10. Endpoint Quick Reference

Module	Inbound Adapter	Outbound Adapter	Inbound Gateway	Outbound Gateway
AMQP	Inbound Channel Adapter	Outbound Channel Adapter	Inbound Gateway	Outbound Gateway
Events	Receiving Spring Application Events	Sending Spring Application Events	N	N
Feed	Feed Inbound Channel Adapter	N	N	N

Module	Inbound Adapter	Outbound Adapter	Inbound Gateway	Outbound Gateway
File	Reading Files and 'tail'ing Files	Writing files	N	Writing files
FTP(S)	FTP Inbound Channel Adapter	FTP Outbound Channel Adapter	N	FTP Outbound Gateway
Gemfire	Inbound Channel Adapter and Continuous Query Inbound Channel Adapter	Outbound Channel Adapter	N	N
HTTP	HTTP Namespace Support	HTTP Namespace Support	Http Inbound Components	HTTP Outbound Components
JDBC	Inbound Channel Adapter and Stored Procedure Inbound Channel Adapter	Outbound Channel Adapter and Stored Procedure Outbound Channel Adapter	N	Outbound Gateway and Stored Procedure Outbound Gateway
JMS	Inbound Channel Adapter and Message-driven Channel Adapter	Outbound Channel Adapter	Inbound Gateway	Outbound Gateway
JMX	Notification-listening Channel Adapter and Attribute-polling Channel Adapter and Tree-polling Channel Adapter	Notification-publishing Channel Adapter and Operation-invoking Channel Adapter	N	Operation-invoking Outbound Gateway
JPA	Inbound Channel Adapter	Outbound Channel Adapter	N	Updating Outbound Gateway and Retrieving Outbound Gateway
Mail	Mail-receiving Channel Adapter	Mail-sending Channel Adapter	N	N
MongoDB	MongoDB Inbound Channel Adapter	MongoDB Outbound Channel Adapter	N	N
MQTT	Inbound (Message-driven) Channel Adapter	Outbound Channel Adapter	N	N

Module	Inbound Adapter	Outbound Adapter	Inbound Gateway	Outbound Gateway
Redis	Redis Inbound Channel Adapter and Redis Queue Inbound Channel Adapter and Redis Store Inbound Channel Adapter	Redis Outbound Channel Adapter and Redis Queue Outbound Channel Adapter and RedisStore Outbound Channel Adapter	Redis Queue Inbound Gateway	Redis Outbound Command Gateway and Redis Queue Outbound Gateway
Resource	Resource Inbound Channel Adapter	N	N	N
RMI	N	N	Inbound RMI	Outbound RMI
RSocket	N	N	RSocket Inbound Gateway	RSocket Outbound Gateway
SFTP	SFTP Inbound Channel Adapter	SFTP Outbound Channel Adapter	N	SFTP Outbound Gateway
STOMP	STOMP Inbound Channel Adapter	STOMP Outbound Channel Adapter	N	N
Stream	Reading from Streams	Writing to Streams	N	N
Syslog	Syslog Inbound Channel Adapter	N	N	N
TCP	TCP Adapters	TCP Adapters	TCP Gateways	TCP Gateways
UDP	UDP Adapters	UDP Adapters	N	N
WebFlux	WebFlux Inbound Channel Adapter	WebFlux Outbound Channel Adapter	Inbound WebFlux Gateway	Outbound WebFlux Gateway
Web Services	N	N	Inbound Web Service Gateways	Outbound Web Service Gateways
Web Sockets	WebSocket Inbound Channel Adapter	WebSocket Outbound Channel Adapter	N	N
XMPP	XMPP Messages and XMPP Presence	XMPP Messages and XMPP Presence	N	N

In addition, as discussed in Core Messaging, Spring Integration provides endpoints for interfacing with Plain Old Java Objects (POJOs). As discussed in Channel Adapter, the `<int:inbound-channel-adapter>` element lets you poll a Java method for data. The `<int:outbound-channel-adapter>` element lets you send data to a `void` method. As discussed in Messaging Gateways, the `<int:gateway>` element lets any Java program invoke a messaging flow. Each of these works without requiring any source-

level dependencies on Spring Integration. The equivalent of an outbound gateway in this context is using a service activator (see Service Activator) to invoke a method that returns an `Object` of some kind.

Starting with version `5.2.2`, all the inbound gateways can be configured with an `errorOnTimeout` boolean flag to throw a `MessageTimeoutException` when the downstream flow doesn't return a reply during the reply timeout. The timer is not started until the thread returns control to the gateway, so usually it is only useful when the downstream flow is asynchronous or it stops because of a `null` return from some handler, e.g. filter. Such an exception can be handled on the `errorChannel` flow, e.g. producing a compensation reply for requesting client.

Chapter 15. AMQP Support

Spring Integration provides channel adapters for receiving and sending messages by using the Advanced Message Queuing Protocol (AMQP).

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-amqp</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-amqp:5.3.0.RC1"
```

The following adapters are available:

- Inbound Channel Adapter
- Inbound Gateway
- Outbound Channel Adapter
- Outbound Gateway
- Async Outbound Gateway

Spring Integration also provides a point-to-point message channel and a publish-subscribe message channel backed by AMQP Exchanges and Queues.

To provide AMQP support, Spring Integration relies on (Spring AMQP), which applies core Spring concepts to the development of AMQP-based messaging solutions. Spring AMQP provides similar semantics to (Spring JMS).

Whereas the provided AMQP Channel Adapters are intended for unidirectional messaging (send or receive) only, Spring Integration also provides inbound and outbound AMQP gateways for request-reply operations.

TIP: You should familiarize yourself with the reference documentation of the Spring AMQP project. It provides much more in-depth information about Spring's integration with AMQP in general and RabbitMQ in particular.

15.1. Inbound Channel Adapter

The following listing shows the possible configuration options for an AMQP Inbound Channel

Adapter:

```
<int-amqp:inbound-channel-adapter
                                  id="inboundAmqp"                  ①
                                  channel="inboundChannel"          ②
                                  queue-names="si.test.queue"       ③
                                  acknowledge-mode="AUTO"           ④
                                  advice-chain=""                   ⑤
                                  channel-transacted=""             ⑥
                                  concurrent-consumers=""           ⑦
                                  connection-factory=""             ⑧
                                  error-channel=""                  ⑨
                                  expose-listener-channel=""        ⑩
                                  header-mapper=""                  ⑪
                                  mapped-request-headers=""         ⑫
                                  listener-container=""             ⑬
                                  message-converter=""              ⑭
                                  message-properties-converter="" ⑮
                                  phase=""                          ⑯
                                  prefetch-count=""                 ⑰
                                  receive-timeout=""                ⑱
                                  recovery-interval=""              ⑲
                                  missing-queues-fatal=""           ⑳
                                  shutdown-timeout=""
                                  task-executor=""
                                  transaction-attribute=""
                                  transaction-manager=""
                                  tx-size=""
                                  consumers-per-queue
                                  batch-mode="MESSAGES"/>
```

① The unique ID for this adapter. Optional.

② Message channel to which converted messages should be sent. Required.

③ Names of the AMQP queues (comma-separated list) from which messages should be consumed. Required.

④ Acknowledge mode for the `MessageListenerContainer`. When set to `MANUAL`, the delivery tag and channel are provided in message headers `amqp_deliveryTag` and `amqp_channel`, respectively. The user application is responsible for acknowledgement. `NONE` means no acknowledgements (`autoAck`). `AUTO` means the adapter's container acknowledges when the downstream flow completes. Optional (defaults to AUTO). See Inbound Endpoint Acknowledge Mode.

⑤ Extra AOP Advices to handle cross-cutting behavior associated with this inbound channel adapter. Optional.

⑥ Flag to indicate that channels created by this component are transactional. If true, it tells the framework to use a transactional channel and to end all operations (send or receive) with a commit or rollback, depending on the outcome, with an exception that signals a rollback. Optional (Defaults to false).

⑦ Specify the number of concurrent consumers to create. The default is 1. We recommend

raising the number of concurrent consumers to scale the consumption of messages coming in from a queue. However, note that any ordering guarantees are lost once multiple consumers are registered. In general, use one consumer for low-volume queues. Not allowed when 'consumers-per-queue' is set. Optional.

⑧ Bean reference to the RabbitMQ `ConnectionFactory`. Optional (defaults to `connectionFactory`).

⑨ Message channel to which error messages should be sent. Optional.

⑩ Whether the listener channel (com.rabbitmq.client.Channel) is exposed to a registered `ChannelAwareMessageListener`. Optional (defaults to true).

⑪ A reference to an `AmqpHeaderMapper` to use when receiving AMQP Messages. Optional. By default, only standard AMQP properties (such as `contentType`) are copied to Spring Integration `MessageHeaders`. Any user-defined headers within the AMQP `MessageProperties` are NOT copied to the message by the default `DefaultAmqpHeaderMapper`. Not allowed if 'request-header-names' is provided.

⑫ Comma-separated list of the names of AMQP Headers to be mapped from the AMQP request into the `MessageHeaders`. This can only be provided if the 'header-mapper' reference is not provided. The values in this list can also be simple patterns to be matched against the header names (such as "*" or "thing1*, thing2" or "*something").

⑬ Reference to the `AbstractMessageListenerContainer` to use for receiving AMQP Messages. If this attribute is provided, no other attribute related to the listener container configuration should be provided. In other words, by setting this reference, you must take full responsibility for the listener container configuration. The only exception is the `MessageListener` itself. Since that is actually the core responsibility of this channel adapter implementation, the referenced listener container must not already have its own `MessageListener`. Optional.

⑭ The `MessageConverter` to use when receiving AMQP messages. Optional.

⑮ The `MessagePropertiesConverter` to use when receiving AMQP messages. Optional.

⑯ Specifies the phase in which the underlying `AbstractMessageListenerContainer` should be started and stopped. The startup order proceeds from lowest to highest, and the shutdown order is the reverse of that. By default, this value is `Integer.MAX_VALUE`, meaning that this container starts as late as possible and stops as soon as possible. Optional.

⑰ Tells the AMQP broker how many messages to send to each consumer in a single request. Often, you can set this value high to improve throughput. It should be greater than or equal to the transaction size (see the `tx-size` attribute, later in this list). Optional (defaults to `1`).

⑱ Receive timeout in milliseconds. Optional (defaults to `1000`).

⑲ Specifies the interval between recovery attempts of the underlying `AbstractMessageListenerContainer` (in milliseconds). Optional (defaults to `5000`).

⑳ If 'true' and none of the queues are available on the broker, the container throws a fatal exception during startup and stops if the queues are deleted when the container is running (after making three attempts to passively declare the queues). If `false`, the container does not throw an exception and goes into recovery mode, attempting to restart according to the `recovery-interval`. Optional (defaults to `true`).

The time to wait for workers (in milliseconds) after the underlying

`AbstractMessageListenerContainer` is stopped and before the AMQP connection is forced closed. If any workers are active when the shutdown signal comes, they are allowed to finish processing as long as they can finish within this timeout. Otherwise, the connection is closed and messages remain unacknowledged (if the channel is transactional). Optional (defaults to `5000`).

By default, the underlying `AbstractMessageListenerContainer` uses a `SimpleAsyncTaskExecutor` implementation, that fires up a new thread for each task, running it asynchronously. By default, the number of concurrent threads is unlimited. Note that this implementation does not reuse threads. Consider using a thread-pooling `TaskExecutor` implementation as an alternative. Optional (defaults to `SimpleAsyncTaskExecutor`).

By default, the underlying `AbstractMessageListenerContainer` creates a new instance of the `DefaultTransactionAttribute` (it takes the EJB approach to rolling back on runtime but not checked exceptions). Optional (defaults to `DefaultTransactionAttribute`).

Sets a bean reference to an external `PlatformTransactionManager` on the underlying `AbstractMessageListenerContainer`. The transaction manager works in conjunction with the `channel-transacted` attribute. If there is already a transaction in progress when the framework is sending or receiving a message and the `channelTransacted` flag is `true`, the commit or rollback of the messaging transaction is deferred until the end of the current transaction. If the `channelTransacted` flag is `false`, no transaction semantics apply to the messaging operation (it is auto-acked). For further information, see Transactions with Spring AMQP. Optional.

Tells the `SimpleMessageListenerContainer` how many messages to process in a single transaction (if the channel is transactional). For best results, it should be less than or equal to the value set in `prefetch-count`. Not allowed when 'consumers-per-queue' is set. Optional (defaults to `1`).

Indicates that the underlying listener container should be a `DirectMessageListenerContainer` instead of the default `SimpleMessageListenerContainer`. See the Spring AMQP Reference Manual for more information.

When the container's `consumerBatchEnabled` is `true`, determines how the adapter presents the batch of messages in the message payload. When set to `MESSAGES` (default), the payload is a `List<Message<?>>` where each message has headers mapped from the incoming AMQP `Message` and the payload is the converted `body`. When set to `EXTRACT_PAYLOADS`, the payload is a `List<?>` where the elements are converted from the AMQP `Message` body.

container

Note that when configuring an external container, you cannot use the Spring AMQP namespace to define the container. This is because the namespace requires at least one `<listener/>` element. In this environment, the listener is internal to the adapter. For this reason, you must define the container by using a normal Spring `<bean/>` definition, as the following example shows:

```
<bean id="container"
 class="org.springframework.amqp.rabbit.listener.SimpleMessageListenerC
ontainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="queueNames" value="aName.queue" />
    <property name="defaultRequeueRejected" value="false"/>
</bean>
```

Even though the Spring Integration JMS and AMQP support is similar, important differences exist. The JMS inbound channel adapter is using a `JmsDestinationPollingSource` under the covers and expects a configured poller. The AMQP inbound channel adapter uses an `AbstractMessageListenerContainer` and is message driven. In that regard, it is more similar to the JMS message-driven channel adapter.

15.1.1. Configuring with Java Configuration

The following Spring Boot application shows an example of configuring the inbound adapter with Java configuration:

```
@SpringBootApplication
public class AmqpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(AmqpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public MessageChannel amqpInputChannel() {
        return new DirectChannel();
    }

    @Bean
    public AmqpInboundChannelAdapter inbound(SimpleMessageListenerContainer
listenerContainer,
            @Qualifier("amqpInputChannel") MessageChannel channel) {
        AmqpInboundChannelAdapter adapter = new AmqpInboundChannelAdapter
(listenerContainer);
        adapter.setOutputChannel(channel);
        return adapter;
    }

    @Bean
    public SimpleMessageListenerContainer container(ConnectionFactory
connectionFactory) {
        SimpleMessageListenerContainer container =
                                    new SimpleMessageListenerContainer
(connectionFactory);
        container.setQueueNames("aName");
        container.setConcurrentConsumers(2);
        // ...
        return container;
    }

    @Bean
    @ServiceActivator(inputChannel = "amqpInputChannel")
    public MessageHandler handler() {
        return new MessageHandler() {

            @Override
            public void handleMessage(Message<?> message) throws
MessagingException {
                System.out.println(message.getPayload());
            }

        };
    }
```

```
}
```

15.1.2. Configuring with the Java DSL

The following Spring Boot application provides an example of configuring the inbound adapter with the Java DSL:

```
@SpringBootApplication
public class AmqpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(AmqpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public IntegrationFlow amqpInbound(ConnectionFactory connectionFactory) {
        return IntegrationFlows.from(Amqp.inboundAdapter(connectionFactory, "
aName"))
                .handle(m -> System.out.println(m.getPayload()))
                .get();
    }

}
```

15.1.3. Batched Messages

See the Spring AMQP Documentation for more information about batched messages.

To produce batched messages with Spring Integration, simply configure the outbound endpoint with a `BatchingRabbitTemplate`.

When receiving batched messages, by default, the listener containers extract each fragment message and the adapter will produce a `Message<?>` for each fragment. Starting with version 5.2, if the container's `deBatchingEnabled` property is set to `false`, the de-batching is performed by the adapter instead, and a single `Message<List<?>>` is produced with the payload being a list of the fragment payloads (after conversion if appropriate).

The default `BatchingStrategy` is the `SimpleBatchingStrategy`, but this can be overridden on the adapter.

15.2. Polled Inbound Channel Adapter

15.2.1. Overview

Version 5.0.1 introduced a polled channel adapter, letting you fetch individual messages on demand — for example, with a `MessageSourcePollingTemplate` or a poller. See Deferred Acknowledgment Pollable Message Source for more information.

It does not currently support XML configuration.

The following example shows how to configure an `AmqpMessageSource` with Java configuration:

```
@Bean
public AmqpMessageSource source(ConnectionFactory connectionFactory) {
    return new AmpqpMessageSource(connectionFactory, "someQueue");
}
```

See the Javadoc for configuration properties.

The following example shows how to configure an `inboundPolledAdapter` with the Java DSL:

```
@Bean
public IntegrationFlow flow() {
    return IntegrationFlows.from(Amqp.inboundPolledAdapter(connectionFactory(),
DSL_QUEUE),
                    e -> e.poller(Pollers.fixedDelay(1_000)).autoStartup(false))
            .handle(p -> {
                ...
            })
            .get();
}
```

15.2.2. Batched Messages

See Batched Messages.

For the polled adapter, there is no listener container, batched messages are always debatched (if the `BatchingStrategy` supports doing so).

15.3. Inbound Gateway

The inbound gateway supports all the attributes on the inbound channel adapter (except that 'channel' is replaced by 'request-channel'), plus some additional attributes. The following listing shows the available attributes:

```
<int-amqp:inbound-gateway
                          id="inboundGateway"                ①
                          request-channel="myRequestChannel" ②
                          header-mapper=""                   ③
                          mapped-request-headers=""          ④
                          mapped-reply-headers=""            ⑤
                          reply-channel="myReplyChannel"     ⑥
                          reply-timeout="1000"               ⑦
                          amqp-template=""                   ⑧
                          default-reply-to="" />             ⑨
```

① The Unique ID for this adapter. Optional.

② Message channel to which converted messages are sent. Required.

③ A reference to an `AmqpHeaderMapper` to use when receiving AMQP Messages. Optional. By default, only standard AMQP properties (such as `contentType`) are copied to and from Spring Integration `MessageHeaders`. Any user-defined headers within the AMQP `MessageProperties` are not copied to or from an AMQP message by the default `DefaultAmqpHeaderMapper`. Not allowed if 'request-header-names' or 'reply-header-names' is provided.

④ Comma-separated list of names of AMQP Headers to be mapped from the AMQP request into the `MessageHeaders`. This attribute can be provided only if the 'header-mapper' reference is not provided. The values in this list can also be simple patterns to be matched against the header names (e.g. `"*"` or `"thing1*, thing2"` or `"*thing1"`).

⑤ Comma-separated list of names of `MessageHeaders` to be mapped into the AMQP message properties of the AMQP reply message. All standard Headers (such as `contentType`) are mapped to AMQP Message Properties, while user-defined headers are mapped to the 'headers' property. This attribute can only be provided if the 'header-mapper' reference is not provided. The values in this list can also be simple patterns to be matched against the header names (for example, `"*"` or `"foo*, bar"` or `"*foo"`).

⑥ Message Channel where reply Messages are expected. Optional.

⑦ Sets the `receiveTimeout` on the underlying `o.s.i.core.MessagingTemplate` for receiving messages from the reply channel. If not specified, this property defaults to `1000` (1 second). Only applies if the container thread hands off to another thread before the reply is sent.

⑧ The customized `AmqpTemplate` bean reference (to have more control over the reply messages to send). You can provide an alternative implementation to the `RabbitTemplate`.

⑨ The `replyTo` `o.s.amqp.core.Address` to be used when the `requestMessage` does not have a `replyTo` property. If this option is not specified, no `amqp-template` is provided, no `replyTo` property exists in the request message, and an `IllegalStateException` is thrown because the reply cannot be routed. If this option is not specified and an external `amqp-template` is provided, no exception is thrown. You must either specify this option or configure a default `exchange` and `routingKey` on that template, if you anticipate cases when no `replyTo` property exists in the request message.

See the note in Inbound Channel Adapter about configuring the `listener-container` attribute.

15.3.1. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the inbound gateway with Java configuration:

```
@SpringBootApplication
public class AmqpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(AmqpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public MessageChannel amqpInputChannel() {
        return new DirectChannel();
    }

    @Bean
    public AmqpInboundGateway inbound(SimpleMessageListenerContainer
listenerContainer,
            @Qualifier("amqpInputChannel") MessageChannel channel) {
        AmqpInboundGateway gateway = new AmqpInboundGateway(listenerContainer);
        gateway.setRequestChannel(channel);
        gateway.setDefaultReplyTo("bar");
        return gateway;
    }

    @Bean
    public SimpleMessageListenerContainer container(ConnectionFactory
connectionFactory) {
        SimpleMessageListenerContainer container =
                        new SimpleMessageListenerContainer(connectionFactory);
        container.setQueueNames("foo");
        container.setConcurrentConsumers(2);
        // ...
        return container;
    }

    @Bean
    @ServiceActivator(inputChannel = "amqpInputChannel")
    public MessageHandler handler() {
        return new AbstractReplyProducingMessageHandler() {

            @Override
            protected Object handleRequestMessage(Message<?> requestMessage) {
                return "reply to " + requestMessage.getPayload();
            }

        };
    }

}
```

15.3.2. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the inbound gateway with the Java DSL:

```
@SpringBootApplication
public class AmqpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(AmqpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean // return the upper cased payload
    public IntegrationFlow amqpInboundGateway(ConnectionFactory connectionFactory)
{
        return IntegrationFlows.from(Amqp.inboundGateway(connectionFactory, "foo"
))
                .transform(String.class, String::toUpperCase)
                .get();
    }

}
```

15.3.3. Batched Messages

See Batched Messages.

15.4. Inbound Endpoint Acknowledge Mode

By default, the inbound endpoints use the `AUTO` acknowledge mode, which means the container automatically acknowledges the message when the downstream integration flow completes (or a message is handed off to another thread by using a `QueueChannel` or `ExecutorChannel`). Setting the mode to `NONE` configures the consumer such that acknowledgments are not used at all (the broker automatically acknowledges the message as soon as it is sent). Setting the mode to `MANUAL` lets user code acknowledge the message at some other point during processing. To support this, with this mode, the endpoints provide the `Channel` and `deliveryTag` in the `amqp_channel` and `amqp_deliveryTag` headers, respectively.

You can perform any valid Rabbit command on the `Channel` but, generally, only `basicAck` and `basicNack` (or `basicReject`) are used. In order to not interfere with the operation of the container, you should not retain a reference to the channel and use it only in the context of the current message.

Since the `Channel` is a reference to a “live” object, it cannot be serialized and is lost if a message is persisted.

The following example shows how you might use `MANUAL` acknowledgement:

```
@ServiceActivator(inputChannel = "foo", outputChannel = "bar")
public Object handle(@Payload String payload, @Header(AmqpHeaders.CHANNEL) Channel
channel,
        @Header(AmqpHeaders.DELIVERY_TAG) Long deliveryTag) throws Exception {

    // Do some processing

    if (allOK) {
        channel.basicAck(deliveryTag, false);

        // perhaps do some more processing

    }
    else {
        channel.basicNack(deliveryTag, false, true);
    }
    return someResultForDownStreamProcessing;
}
```

15.5. Outbound Channel Adapter

The following outbound endpoints have many similar configuration options. Starting with version 5.2, the `confirm-timeout` has been added. Normally, when publisher confirms are enabled, the broker will quickly return an ack (or nack) which will be sent to the appropriate channel. If a channel is closed before the confirm is received, the Spring AMQP framework will synthesize a nack. "Missing" acks should never occur but, if you set this property, the endpoint will periodically check for them and synthesize a nack if the time elapses without a confirm being received.

15.6. Outbound Channel Adapter

The following example shows the available properties for an AMQP outbound channel adapter:

```
<int-amqp:outbound-channel-adapter id="outboundAmqp"                ①
                                   channel="outboundChannel"        ②
                                   amqp-template="myAmqpTemplate"   ③
                                   exchange-name=""                 ④
                                   exchange-name-expression=""      ⑤
                                   order="1"                        ⑥
                                   routing-key=""                   ⑦
                                   routing-key-expression=""        ⑧
                                   default-delivery-mode""          ⑨
                                   confirm-correlation-expression="" ⑩
                                   confirm-ack-channel=""           ⑪
                                   confirm-nack-channel=""          ⑫
                                   confirm-timeout=""               ⑬
                                   wait-for-confirm=""              ⑭
                                   return-channel=""                ⑮
                                   error-message-strategy=""        ⑯
                                   header-mapper=""                 ⑰
                                   mapped-request-headers=""        ⑱
                                   lazy-connect="true"              ⑲
                                   multi-send="false"/>             ⑳
```

① The unique ID for this adapter. Optional.

② Message channel to which messages should be sent to have them converted and published to an AMQP exchange. Required.

③ Bean reference to the configured AMQP template. Optional (defaults to `amqpTemplate`).

④ The name of the AMQP exchange to which messages are sent. If not provided, messages are sent to the default, no-name exchange. Mutually exclusive with 'exchange-name-expression'. Optional.

⑤ A SpEL expression that is evaluated to determine the name of the AMQP exchange to which messages are sent, with the message as the root object. If not provided, messages are sent to the default, no-name exchange. Mutually exclusive with 'exchange-name'. Optional.

⑥ The order for this consumer when multiple consumers are registered, thereby enabling load-balancing and failover. Optional (defaults to `Ordered.LOWEST_PRECEDENCE [=Integer.MAX_VALUE]`).

⑦ The fixed routing-key to use when sending messages. By default, this is an empty `String`. Mutually exclusive with 'routing-key-expression'. Optional.

⑧ A SpEL expression that is evaluated to determine the routing key to use when sending messages, with the message as the root object (for example, 'payload.key'). By default, this is an empty `String`. Mutually exclusive with 'routing-key'. Optional.

⑨ The default delivery mode for messages: `PERSISTENT` or `NON_PERSISTENT`. Overridden if the `header-mapper` sets the delivery mode. If the Spring Integration message header `amqp_deliveryMode` is present, the `DefaultHeaderMapper` sets the value. If this attribute is not supplied and the header mapper does not set it, the default depends on the underlying Spring AMQP `MessagePropertiesConverter` used by the `RabbitTemplate`. If that is not

customized at all, the default is `PERSISTENT`. Optional.

⑩ An expression that defines correlation data. When provided, this configures the underlying AMQP template to receive publisher confirmations. Requires a dedicated `RabbitTemplate` and a `CachingConnectionFactory` with the `publisherConfirms` property set to `true`. When a publisher confirmation is received and correlation data is supplied, it is written to either the `confirm-ack-channel` or the `confirm-nack-channel`, depending on the confirmation type. The payload of the confirmation is the correlation data, as defined by this expression. The message has an 'amqp_publishConfirm' header set to `true` (`ack`) or `false` (`nack`). Examples: `headers['myCorrelationData']` and `payload`. Version 4.1 introduced the `amqp_publishConfirmNackCause` message header. It contains the `cause` of a 'nack' for a publisher confirmation. Starting with version 4.2, if the expression resolves to a `Message<?>` instance (such as `#this`), the message emitted on the `ack`/`nack` channel is based on that message, with the additional header(s) added. Previously, a new message was created with the correlation data as its payload, regardless of type. Optional.

⑪ The channel to which positive (`ack`) publisher confirms are sent. The payload is the correlation data defined by the `confirm-correlation-expression`. If the expression is `#root` or `#this`, the message is built from the original message, with the `amqp_publishConfirm` header set to `true`. Optional (the default is `nullChannel`).

⑫ The channel to which negative (`nack`) publisher confirmations are sent. The payload is the correlation data defined by the `confirm-correlation-expression` (if there is no `ErrorMessageStrategy` configured). If the expression is `#root` or `#this`, the message is built from the original message, with the `amqp_publishConfirm` header set to `false`. When there is an `ErrorMessageStrategy`, the message is an `ErrorMessage` with a `NackedAmqpMessageException` payload. Optional (the default is `nullChannel`).

⑬ When set, the adapter will synthesize a negative acknowledgment (nack) if a publisher confirm is not received within this time in milliseconds. Pending confirms are checked every 50% of this value, so the actual time a nack is sent will be between 1x and 1.5x this value. Default none (nacks will not be generated).

⑭ When set to true, the calling thread will block, waiting for a publisher confirmation. This requires a `RabbitTemplate` configured for confirms as well as a `confirm-correlation-expression`. The thread will block for up to `confirm-timeout` (or 5 seconds by default). If a timeout occurs, a `MessageTimeoutException` will be thrown. If returns are enabled and a message is returned, or any other exception occurs while awaiting the confirm, a `MessageHandlingException` will be thrown, with an appropriate message.

⑮ The channel to which returned messages are sent. When provided, the underlying AMQP template is configured to return undeliverable messages to the adapter. When there is no `ErrorMessageStrategy` configured, the message is constructed from the data received from AMQP, with the following additional headers: `amqp_returnReplyCode`, `amqp_returnReplyText`, `amqp_returnExchange`, `amqp_returnRoutingKey`. When there is an `ErrorMessageStrategy`, the message is an `ErrorMessage` with a `ReturnedAmqpMessageException` payload. Optional.

⑯ A reference to an `ErrorMessageStrategy` implementation used to build `ErrorMessage` instances when sending returned or negatively acknowledged messages.

⑰ A reference to an `AmqpHeaderMapper` to use when sending AMQP Messages. By default, only standard AMQP properties (such as `contentType`) are copied to the Spring Integration `MessageHeaders`. Any user-defined headers is not copied to the message by the

default`DefaultAmqpHeaderMapper`. Not allowed if 'request-header-names' is provided. Optional.

⑱ Comma-separated list of names of AMQP Headers to be mapped from the `MessageHeaders` to the AMQP Message. Not allowed if the 'header-mapper' reference is provided. The values in this list can also be simple patterns to be matched against the header names (e.g. `"*"` or `"thing1*, thing2"` or `"*thing1"`).

⑲ When set to `false`, the endpoint attempts to connect to the broker during application context initialization. This allows “fail fast” detection of bad configuration but also causes initialization to fail if the broker is down. When `true` (the default), the connection is established (if it does not already exist because some other component established it) when the first message is sent.

⑳ When set to `true`, payloads of type `Iterable<Message<?>>` will be sent as discrete messages on the same channel within the scope of a single `RabbitTemplate` invocation. Requires a `RabbitTemplate`. When `wait-for-confirms` is true, `RabbitTemplate.waitForConfirmsOrDie()` is invoked after the messages have been sent. With a transactional template, the sends will be performed in either a new transaction or one that has already been started (if present).

return-channel

Using a `return-channel` requires a `RabbitTemplate` with the `mandatory` property set to `true` and a `CachingConnectionFactory` with the `publisherReturns` property set to `true`. When using multiple outbound endpoints with returns, a separate `RabbitTemplate` is needed for each endpoint.

15.6.1. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the outbound adapter with Java configuration:

```
@SpringBootApplication
@IntegrationComponentScan
public class AmqpJavaApplication {

    public static void main(String[] args) {
         ConfigurableApplicationContext context =
              new SpringApplicationBuilder(AmqpJavaApplication.class)
                       .web(false)
                       .run(args);
         MyGateway gateway = context.getBean(MyGateway.class);
         gateway.sendToRabbit("foo");
    }

    @Bean
    @ServiceActivator(inputChannel = "amqpOutboundChannel")
    public AmqpOutboundEndpoint amqpOutbound(AmqpTemplate amqpTemplate) {
        AmqpOutboundEndpoint outbound = new AmqpOutboundEndpoint(amqpTemplate);
        outbound.setRoutingKey("foo"); // default exchange - route to queue 'foo'
        return outbound;
    }

    @Bean
    public MessageChannel amqpOutboundChannel() {
        return new DirectChannel();
    }

    @MessagingGateway(defaultRequestChannel = "amqpOutboundChannel")
    public interface MyGateway {

        void sendToRabbit(String data);

    }

}
```

15.6.2. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the outbound adapter with the Java DSL:

```
@SpringBootApplication
@IntegrationComponentScan
public class AmqpJavaApplication {

    public static void main(String[] args) {
         ConfigurableApplicationContext context =
                  new SpringApplicationBuilder(AmqpJavaApplication.class)
                          .web(false)
                          .run(args);
         MyGateway gateway = context.getBean(MyGateway.class);
         gateway.sendToRabbit("foo");
    }

    @Bean
    public IntegrationFlow amqpOutbound(AmqpTemplate amqpTemplate) {
        return IntegrationFlows.from(amqpOutboundChannel())
                .handle(Amqp.outboundAdapter(amqpTemplate)
                            .routingKey("foo")) // default exchange - route to
queue 'foo'
                .get();
    }

    @Bean
    public MessageChannel amqpOutboundChannel() {
        return new DirectChannel();
    }

    @MessagingGateway(defaultRequestChannel = "amqpOutboundChannel")
    public interface MyGateway {

        void sendToRabbit(String data);

    }
}
```

15.7. Outbound Gateway

The following listing shows the possible properties for an AMQP Outbound Gateway:

```
<int-amqp:outbound-gateway id="outboundGateway"                ①
                           request-channel="myRequestChannel" ②
                           amqp-template=""                   ③
                           exchange-name=""                   ④
                           exchange-name-expression=""        ⑤
                           order="1"                          ⑥
                           reply-channel=""                   ⑦
                           reply-timeout=""                   ⑧
                           requires-reply=""                  ⑨
                           routing-key=""                     ⑩
                           routing-key-expression=""          ⑪
                           default-delivery-mode""            ⑫
                           confirm-correlation-expression=""  ⑬
                           confirm-ack-channel=""             ⑭
                           confirm-nack-channel=""            ⑮
                           confirm-timeout=""                 ⑯
                           return-channel=""                  ⑰
                           error-message-strategy=""          ⑱
                           lazy-connect="true" />             ⑲
```

① The unique ID for this adapter. Optional.

② Message channel to which messages are sent to have them converted and published to an AMQP exchange. Required.

③ Bean reference to the configured AMQP template. Optional (defaults to `amqpTemplate`).

④ The name of the AMQP exchange to which messages should be sent. If not provided, messages are sent to the default, no-name cxchange. Mutually exclusive with 'exchange-name-expression'. Optional.

⑤ A SpEL expression that is evaluated to determine the name of the AMQP exchange to which messages should be sent, with the message as the root object. If not provided, messages are sent to the default, no-name exchange. Mutually exclusive with 'exchange-name'. Optional.

⑥ The order for this consumer when multiple consumers are registered, thereby enabling load-balancing and failover. Optional (defaults to `Ordered.LOWEST_PRECEDENCE [=Integer.MAX_VALUE]`).

⑦ Message channel to which replies should be sent after being received from an AMQP queue and converted. Optional.

⑧ The time the gateway waits when sending the reply message to the `reply-channel`. This only applies if the `reply-channel` can block — such as a `QueueChannel` with a capacity limit that is currently full. Defaults to infinity.

⑨ When `true`, the gateway throws an exception if no reply message is received within the `AmqpTemplate`s `replyTimeout` property. Defaults to `true`.

⑩ The `routing-key` to use when sending messages. By default, this is an empty `String`. Mutually exclusive with 'routing-key-expression'. Optional.

⑪ A SpEL expression that is evaluated to determine the `routing-key` to use when sending

messages, with the message as the root object (for example, 'payload.key'). By default, this is an empty `String`. Mutually exclusive with 'routing-key'. Optional.

⑫ The default delivery mode for messages: `PERSISTENT` or `NON_PERSISTENT`. Overridden if the `header-mapper` sets the delivery mode. If the Spring Integration message header `amqp_deliveryMode` is present, the `DefaultHeaderMapper` sets the value. If this attribute is not supplied and the header mapper does not set it, the default depends on the underlying Spring AMQP `MessagePropertiesConverter` used by the `RabbitTemplate`. If that is not customized at all, the default is `PERSISTENT`. Optional.

⑬ Since version 4.2. An expression defining correlation data. When provided, this configures the underlying AMQP template to receive publisher confirms. Requires a dedicated `RabbitTemplate` and a `CachingConnectionFactory` with the `publisherConfirms` property set to `true`. When a publisher confirm is received and correlation data is supplied, it is written to either the `confirm-ack-channel` or the `confirm-nack-channel`, depending on the confirmation type. The payload of the confirm is the correlation data, as defined by this expression. The message has a header 'amqp_publishConfirm' set to `true` (`ack`) or `false` (`nack`). For `nack` confirmations, Spring Integration provides an additional header `amqp_publishConfirmNackCause`. Examples: `headers['myCorrelationData']` and `payload`. If the expression resolves to a `Message<?>` instance (such as `#this`), the message emitted on the `ack` /`nack` channel is based on that message, with the additional headers added. Previously, a new message was created with the correlation data as its payload, regardless of type. Optional.

⑭ The channel to which positive (`ack`) publisher confirmations are sent. The payload is the correlation data defined by `confirm-correlation-expression`. If the expression is `#root` or `#this`, the message is built from the original message, with the `amqp_publishConfirm` header set to `true`. Optional (the default is `nullChannel`).

⑮ The channel to which negative (`nack`) publisher confirmations are sent. The payload is the correlation data defined by `confirm-correlation-expression` (if there is no `ErrorMessageStrategy` configured). If the expression is `#root` or `#this`, the message is built from the original message, with the `amqp_publishConfirm` header set to `false`. When there is an `ErrorMessageStrategy`, the message is an `ErrorMessage` with a `NackedAmqpMessageException` payload. Optional (the default is `nullChannel`).

⑯ When set, the gateway will synthesize a negative acknowledgment (nack) if a publisher confirm is not received within this time in milliseconds. Pending confirms are checked every 50% of this value, so the actual time a nack is sent will be between 1x and 1.5x this value. Default none (nacks will not be generated).

⑰ The channel to which returned messages are sent. When provided, the underlying AMQP template is configured to return undeliverable messages to the adapter. When there is no `ErrorMessageStrategy` configured, the message is constructed from the data received from AMQP, with the following additional headers: `amqp_returnReplyCode`, `amqp_returnReplyText`, `amqp_returnExchange`, and `amqp_returnRoutingKey`. When there is an `ErrorMessageStrategy`, the message is an `ErrorMessage` with a `ReturnedAmqpMessageException` payload. Optional.

⑱ A reference to an `ErrorMessageStrategy` implementation used to build `ErrorMessage` instances when sending returned or negatively acknowledged messages.

⑲ When set to `false`, the endpoint attempts to connect to the broker during application context initialization. This allows “fail fast” detection of bad configuration by logging an

error message if the broker is down. When `true` (the default), the connection is established (if it does not already exist because some other component established it) when the first message is sent.

return-channel

Using a `return-channel` requires a `RabbitTemplate` with the `mandatory` property set to `true` and a `CachingConnectionFactory` with the `publisherReturns` property set to `true`. When using multiple outbound endpoints with returns, a separate `RabbitTemplate` is needed for each endpoint.

The underlying `AmqpTemplate` has a default `replyTimeout` of five seconds. If you require a longer timeout, you must configure it on the `template`.

15.7.1. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the outbound gateway with Java configuration:

```
@SpringBootApplication
@IntegrationComponentScan
public class AmqpJavaApplication {

    public static void main(String[] args) {
         ConfigurableApplicationContext context =
                new SpringApplicationBuilder(AmqpJavaApplication.class)
                       .web(false)
                       .run(args);
         MyGateway gateway = context.getBean(MyGateway.class);
         String reply = gateway.sendToRabbit("foo");
         System.out.println(reply);
    }

    @Bean
    @ServiceActivator(inputChannel = "amqpOutboundChannel")
    public AmqpOutboundEndpoint amqpOutbound(AmqpTemplate amqpTemplate) {
        AmqpOutboundEndpoint outbound = new AmqpOutboundEndpoint(amqpTemplate);
        outbound.setExpectReply(true);
        outbound.setRoutingKey("foo"); // default exchange - route to queue 'foo'
        return outbound;
    }

    @Bean
    public MessageChannel amqpOutboundChannel() {
        return new DirectChannel();
    }

    @MessagingGateway(defaultRequestChannel = "amqpOutboundChannel")
    public interface MyGateway {

        String sendToRabbit(String data);

    }

}
```

Note that the only difference between the outbound adapter and outbound gateway configuration is the setting of the `expectReply` property.

15.7.2. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the outbound adapter with the Java DSL:

```
@SpringBootApplication
@IntegrationComponentScan
public class AmqpJavaApplication {

    public static void main(String[] args) {
         ConfigurableApplicationContext context =
                 new SpringApplicationBuilder(AmqpJavaApplication.class)
                      .web(false)
                      .run(args);
         RabbitTemplate template = context.getBean(RabbitTemplate.class);
         MyGateway gateway = context.getBean(MyGateway.class);
         String reply = gateway.sendToRabbit("foo");
         System.out.println(reply);
    }

    @Bean
    public IntegrationFlow amqpOutbound(AmqpTemplate amqpTemplate) {
        return IntegrationFlows.from(amqpOutboundChannel())
                .handle(Amqp.outboundGateway(amqpTemplate)
                        .routingKey("foo")) // default exchange - route to queue
'foo'
                .get();
    }

    @Bean
    public MessageChannel amqpOutboundChannel() {
        return new DirectChannel();
    }

    @MessagingGateway(defaultRequestChannel = "amqpOutboundChannel")
    public interface MyGateway {

        String sendToRabbit(String data);

    }
}
```

15.8. Asynchronous Outbound Gateway

The gateway discussed in the previous section is synchronous, in that the sending thread is suspended until a reply is received (or a timeout occurs). Spring Integration version 4.3 added an asynchronous gateway, which uses the `AsyncRabbitTemplate` from Spring AMQP. When a message is sent, the thread returns immediately after the send operation completes, and, when the message is received, the reply is sent on the template's listener container thread. This can be useful when the gateway is invoked on a poller thread. The thread is released and is available for other tasks in the framework.

The following listing shows the possible configuration options for an AMQP asynchronous outbound gateway:

```
<int-amqp:outbound-async-gateway id="asyncOutboundGateway"     ①
                            request-channel="myRequestChannel" ②
                            async-template=""                  ③
                            exchange-name=""                   ④
                            exchange-name-expression=""        ⑤
                            order="1"                          ⑥
                            reply-channel=""                   ⑦
                            reply-timeout=""                   ⑧
                            requires-reply=""                  ⑨
                            routing-key=""                     ⑩
                            routing-key-expression=""          ⑪
                            default-delivery-mode""            ⑫
                            confirm-correlation-expression=""  ⑬
                            confirm-ack-channel=""             ⑭
                            confirm-nack-channel=""            ⑮
                            confirm-timeout=""                 ⑯
                            return-channel=""                  ⑰
                            lazy-connect="true" />             ⑱
```

① The unique ID for this adapter. Optional.

② Message channel to which messages should be sent in order to have them converted and published to an AMQP exchange. Required.

③ Bean reference to the configured `AsyncRabbitTemplate`. Optional (it defaults to `asyncRabbitTemplate`).

④ The name of the AMQP exchange to which messages should be sent. If not provided, messages are sent to the default, no-name exchange. Mutually exclusive with 'exchange-name-expression'. Optional.

⑤ A SpEL expression that is evaluated to determine the name of the AMQP exchange to which messages are sent, with the message as the root object. If not provided, messages are sent to the default, no-name exchange. Mutually exclusive with 'exchange-name'. Optional.

⑥ The order for this consumer when multiple consumers are registered, thereby enabling load-balancing and failover. Optional (it defaults to `Ordered.LOWEST_PRECEDENCE [=Integer.MAX_VALUE]`).

⑦ Message channel to which replies should be sent after being received from an AMQP queue and converted. Optional.

⑧ The time the gateway waits when sending the reply message to the `reply-channel`. This only applies if the `reply-channel` can block — such as a `QueueChannel` with a capacity limit that is currently full. The default is infinity.

⑨ When no reply message is received within the `AsyncRabbitTemplate`'s `receiveTimeout` property and this setting is `true`, the gateway sends an error message to the inbound message's `errorChannel` header. When no reply message is received within the `AsyncRabbitTemplate`'s `receiveTimeout` property and this setting is `false`, the gateway sends an error message to the default `errorChannel` (if available). It defaults to `true`.

⑩ The routing-key to use when sending Messages. By default, this is an empty `String`. Mutually

exclusive with 'routing-key-expression'. Optional.

⑪ A SpEL expression that is evaluated to determine the routing-key to use when sending messages, with the message as the root object (for example, 'payload.key'). By default, this is an empty `String`. Mutually exclusive with 'routing-key'. Optional.

⑫ The default delivery mode for messages: `PERSISTENT` or `NON_PERSISTENT`. Overridden if the `header-mapper` sets the delivery mode. If the Spring Integration message header (`amqp_deliveryMode`) is present, the `DefaultHeaderMapper` sets the value. If this attribute is not supplied and the header mapper does not set it, the default depends on the underlying Spring AMQP `MessagePropertiesConverter` used by the `RabbitTemplate`. If that is not customized, the default is `PERSISTENT`. Optional.

⑬ An expression that defines correlation data. When provided, this configures the underlying AMQP template to receive publisher confirmations. Requires a dedicated `RabbitTemplate` and a `CachingConnectionFactory` with its `publisherConfirms` property set to `true`. When a publisher confirmation is received and correlation data is supplied, the confirmation is written to either the `confirm-ack-channel` or the `confirm-nack-channel`, depending on the confirmation type. The payload of the confirmation is the correlation data as defined by this expression, and the message has its 'amqp_publishConfirm' header set to `true` (`ack`) or `false` (`nack`). For `nack` instances, an additional header (`amqp_publishConfirmNackCause`) is provided. Examples: `headers['myCorrelationData']`, `payload`. If the expression resolves to a `Message<?>` instance (such as “#this”), the message emitted on the `ack`/`nack` channel is based on that message, with the additional headers added. Optional.

⑭ The channel to which positive (`ack`) publisher confirmations are sent. The payload is the correlation data defined by the `confirm-correlation-expression`. Requires the underlying `AsyncRabbitTemplate` to have its `enableConfirms` property set to `true`. Optional (the default is `nullChannel`).

⑮ Since version 4.2. The channel to which negative (`nack`) publisher confirmations are sent. The payload is the correlation data defined by the `confirm-correlation-expression`. Requires the underlying `AsyncRabbitTemplate` to have its `enableConfirms` property set to `true`. Optional (the default is `nullChannel`).

⑯ When set, the gateway will synthesize a negative acknowledgment (nack) if a publisher confirm is not received within this time in milliseconds. Pending confirms are checked every 50% of this value, so the actual time a nack is sent will be between 1x and 1.5x this value. Default none (nacks will not be generated).

⑰ The channel to which returned messages are sent. When provided, the underlying AMQP template is configured to return undeliverable messages to the gateway. The message is constructed from the data received from AMQP, with the following additional headers: `amqp_returnReplyCode`, `amqp_returnReplyText`, `amqp_returnExchange`, and `amqp_returnRoutingKey`. Requires the underlying `AsyncRabbitTemplate` to have its `mandatory` property set to `true`. Optional.

⑱ When set to `false`, the endpoint tries to connect to the broker during application context initialization. Doing so allows “fail fast” detection of bad configuration, by logging an error message if the broker is down. When `true` (the default), the connection is established (if it does not already exist because some other component established it) when the first message is sent.

See also Asynchronous Service Activator for more information.

RabbitTemplate

When you use confirmations and returns, we recommend that the `RabbitTemplate` wired into the `AsyncRabbitTemplate` be dedicated. Otherwise, unexpected side-effects may be encountered.

15.8.1. Configuring with Java Configuration

The following configuration shows an example of how to configure the outbound gateway with Java configuration:

```
@Configuration
public class AmqpAsyncConfig {

    @Bean
    @ServiceActivator(inputChannel = "amqpOutboundChannel")
    public AsyncAmqpOutboundGateway amqpOutbound(AmqpTemplate asyncTemplate) {
        AsyncAmqpOutboundGateway outbound = new AsyncAmqpOutboundGateway
(asyncTemplate);
        outbound.setRoutingKey("foo"); // default exchange - route to queue 'foo'
        return outbound;
    }

    @Bean
    public AsyncRabbitTemplate asyncTemplate(RabbitTemplate rabbitTemplate,
                    SimpleMessageListenerContainer replyContainer) {
        return new AsyncRabbitTemplate(rabbitTemplate, replyContainer);
    }

    @Bean
    public SimpleMessageListenerContainer replyContainer() {
        SimpleMessageListenerContainer container = new
SimpleMessageListenerContainer(ccf);
        container.setQueueNames("asyncRQ1");
        return container;
    }

    @Bean
    public MessageChannel amqpOutboundChannel() {
        return new DirectChannel();
    }

}
```

15.8.2. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the outbound adapter with the Java DSL:

```
@SpringBootApplication
public class AmqpAsyncApplication {

    public static void main(String[] args) {
         ConfigurableApplicationContext context =
                 new SpringApplicationBuilder(AmqpAsyncApplication.class)
                      .web(false)
                      .run(args);
         MyGateway gateway = context.getBean(MyGateway.class);
         String reply = gateway.sendToRabbit("foo");
         System.out.println(reply);
    }

    @Bean
    public IntegrationFlow asyncAmqpOutbound(AsyncRabbitTemplate
asyncRabbitTemplate) {
        return f -> f
                .handle(Amqp.asyncOutboundGateway(asyncRabbitTemplate)
                        .routingKey("foo")); // default exchange - route to queue
'foo'
    }

    @MessagingGateway(defaultRequestChannel = "asyncAmqpOutbound.input")
    public interface MyGateway {

        String sendToRabbit(String data);

    }

}
```

15.9. Inbound Message Conversion

Inbound messages, arriving at the channel adapter or gateway, are converted to the `spring-messaging Message<?>` payload using a message converter. By default, a `SimpleMessageConverter` is used, which handles java serialization and text. Headers are mapped using the `DefaultHeaderMapper.inboundMapper()` by default. If a conversion error occurs, and there is no error channel defined, the exception is thrown to the container and handled by the listener container's error handler. The default error handler treats conversion errors as fatal and the message will be rejected (and routed to a dead-letter exchange, if the queue is so configured). If an error channel is defined, the `ErrorMessage` payload is a `ListenerExecutionFailedException` with properties `failedMessage` (the Spring AMQP message that could not be converted) and the `cause`. If the

container `AcknowledgeMode` is `AUTO` (the default) and the error flow consumes the error without throwing an exception, the original message will be acknowledged. If the error flow throws an exception, the exception type, in conjunction with the container's error handler, will determine whether or not the message is requeued. If the container is configured with `AcknowledgeMode.MANUAL`, the payload is a `ManualAckListenerExecutionFailedException` with additional properties `channel` and `deliveryTag`. This enables the error flow to call `basicAck` or `basicNack` (or `basicReject`) for the message, to control its disposition.

15.10. Outbound Message Conversion

Spring AMQP 1.4 introduced the `ContentTypeDelegatingMessageConverter`, where the actual converter is selected based on the incoming content type message property. This can be used by inbound endpoints.

As of Spring Integration version 4.3, you can use the `ContentTypeDelegatingMessageConverter` on outbound endpoints as well, with the `contentType` header specifying which converter is used.

The following example configures a `ContentTypeDelegatingMessageConverter`, with the default converter being the `SimpleMessageConverter` (which handles Java serialization and plain text), together with a JSON converter:

```
<amqp:outbound-channel-adapter id="withContentTypeConverter" channel=
"ctRequestChannel"
                               exchange-name="someExchange"
                               routing-key="someKey"
                               amqp-template="amqpTemplateContentTypeConverter" />

<int:channel id="ctRequestChannel"/>

<rabbit:template id="amqpTemplateContentTypeConverter"
        connection-factory="connectionFactory" message-converter="ctConverter" />

<bean id="ctConverter"
        class="o.s.amqp.support.converter.ContentTypeDelegatingMessageConverter">
    <property name="delegates">
        <map>
            <entry key="application/json">
                <bean class=
"o.s.amqp.support.converter.Jackson2JsonMessageConverter" />
            </entry>
        </map>
    </property>
</bean>
```

Sending a message to `ctRequestChannel` with the `contentType` header set to `application/json` causes the JSON converter to be selected.

This applies to both the outbound channel adapter and gateway.

Starting with version 5.0, headers that are added to the `MessageProperties` of the outbound message are never overwritten by mapped headers (by default). Previously, this was only the case if the message converter was a `ContentTypeDelegatingMessageConverter` (in that case, the header was mapped first so that the proper converter could be selected). For other converters, such as the `SimpleMessageConverter`, mapped headers overwrote any headers added by the converter. This caused problems when an outbound message had some leftover `contentType` headers (perhaps from an inbound channel adapter) and the correct outbound `contentType` was incorrectly overwritten. The work-around was to use a header filter to remove the header before sending the message to the outbound endpoint.

There are, however, cases where the previous behavior is desired — for example, when a `String` payload that contains JSON, the `SimpleMessageConverter` is not aware of the content and sets the `contentType` message property to `text/plain` but your application would like to override that to `application/json` by setting the `contentType` header of the message sent to the outbound endpoint. The `ObjectToJsonTransformer` does exactly that (by default).

There is now a property called `headersMappedLast` on the outbound channel adapter and gateway (as well as on AMQP-backed channels). Setting this to `true` restores the behavior of overwriting the property added by the converter.

Starting with version 5.1.9, a similar `replyHeadersMappedLast` is provided for the `AmqpInboundGateway` when we produce a reply and would like to override headers populated by the converter. See its JavaDocs for more information.

15.11. Outbound User ID

Spring AMQP version 1.6 introduced a mechanism to allow the specification of a default user ID for outbound messages. It has always been possible to set the `AmqpHeaders.USER_ID` header, which now takes precedence over the default. This might be useful to message recipients. For inbound messages, if the message publisher sets the property, it is made available in the `AmqpHeaders.RECEIVED_USER_ID` header. Note that RabbitMQ validates that the user ID is the actual user ID for the connection or that the connection allows impersonation.

To configure a default user ID for outbound messages, configure it on a `RabbitTemplate` and configure the outbound adapter or gateway to use that template. Similarly, to set the user ID property on replies, inject an appropriately configured template into the inbound gateway. See the Spring AMQP documentation for more information.

15.12. Delayed Message Exchange

Spring AMQP supports the RabbitMQ Delayed Message Exchange Plugin. For inbound messages, the `x-delay` header is mapped to the `AmqpHeaders.RECEIVED_DELAY` header. Setting the `AMQPHeaders.DELAY` header causes the corresponding `x-delay` header to be set in outbound messages. You can also

specify the `delay` and `delayExpression` properties on outbound endpoints (`delay-expression` when using XML configuration). These properties take precedence over the `AmqpHeaders.DELAY` header.

15.13. AMQP-backed Message Channels

There are two message channel implementations available. One is point-to-point, and the other is publish-subscribe. Both of these channels provide a wide range of configuration attributes for the underlying `AmqpTemplate` and `SimpleMessageListenerContainer` (as shown earlier in this chapter for the channel adapters and gateways). However, the examples we show here have minimal configuration. Explore the XML schema to view the available attributes.

A point-to-point channel might look like the following example:

```
<int-amqp:channel id="p2pChannel"/>
```

Under the covers, the preceding example causes a `Queue` named `si.p2pChannel` to be declared, and this channel sends to that `Queue` (technically, by sending to the no-name direct exchange with a routing key that matches the name of this `Queue`). This channel also registers a consumer on that `Queue`. If you want the channel to be "pollable" instead of message-driven, provide the `message-driven` flag with a value of `false`, as the following example shows:

```
<int-amqp:channel id="p2pPollableChannel"  message-driven="false"/>
```

A publish-subscribe channel might look like the following:

```
<int-amqp:publish-subscribe-channel id="pubSubChannel"/>
```

Under the covers, the preceding example causes a fanout exchange named `si.fanout.pubSubChannel` to be declared, and this channel sends to that fanout exchange. This channel also declares a server-named exclusive, auto-delete, non-durable `Queue` and binds that to the fanout exchange while registering a consumer on that `Queue` to receive messages. There is no "pollable" option for a publish-subscribe-channel. It must be message-driven.

Starting with version 4.1, AMQP-backed message channels (in conjunction with `channel-transacted`) support `template-channel-transacted` to separate `transactional` configuration for the `AbstractMessageListenerContainer` and for the `RabbitTemplate`. Note that, previously, `channel-transacted` was `true` by default. Now, by default, it is `false` for the `AbstractMessageListenerContainer`.

Prior to version 4.3, AMQP-backed channels only supported messages with `Serializable` payloads and headers. The entire message was converted (serialized) and sent to RabbitMQ. Now, you can set the `extract-payload` attribute (or `setExtractPayload()` when using Java configuration) to `true`. When

this flag is `true`, the message payload is converted and the headers are mapped, in a manner similar to when you use channel adapters. This arrangement lets AMQP-backed channels be used with non-serializable payloads (perhaps with another message converter, such as the `Jackson2JsonMessageConverter`). See AMQP Message Headers for more about the default mapped headers. You can modify the mapping by providing custom mappers that use the `outbound-header-mapper` and `inbound-header-mapper` attributes. You can now also specify a `default-delivery-mode`, which is used to set the delivery mode when there is no `amqp_deliveryMode` header. By default, Spring AMQP `MessageProperties` uses `PERSISTENT` delivery mode.

As with other persistence-backed channels, AMQP-backed channels are intended to provide message persistence to avoid message loss. They are not intended to distribute work to other peer applications. For that purpose, use channel adapters instead.

Starting with version 5.0, the pollable channel now blocks the poller thread for the specified `receiveTimeout` (the default is 1 second). Previously, unlike other `PollableChannel` implementations, the thread returned immediately to the scheduler if no message was available, regardless of the receive timeout. Blocking is a little more expensive than using a `basicGet()` to retrieve a message (with no timeout), because a consumer has to be created to receive each message. To restore the previous behavior, set the poller's `receiveTimeout` to 0.

15.13.1. Configuring with Java Configuration

The following example shows how to configure the channels with Java configuration:

```
@Bean
public AmqpChannelFactoryBean pollable(ConnectionFactory connectionFactory) {
    AmqpChannelFactoryBean factoryBean = new AmqpChannelFactoryBean();
    factoryBean.setConnectionFactory(connectionFactory);
    factoryBean.setQueueName("foo");
    factoryBean.setPubSub(false);
    return factoryBean;
}

@Bean
public AmqpChannelFactoryBean messageDriven(ConnectionFactory connectionFactory) {
    AmqpChannelFactoryBean factoryBean = new AmqpChannelFactoryBean(true);
    factoryBean.setConnectionFactory(connectionFactory);
    factoryBean.setQueueName("bar");
    factoryBean.setPubSub(false);
    return factoryBean;
}

@Bean
public AmqpChannelFactoryBean pubSub(ConnectionFactory connectionFactory) {
    AmqpChannelFactoryBean factoryBean = new AmqpChannelFactoryBean(true);
    factoryBean.setConnectionFactory(connectionFactory);
    factoryBean.setQueueName("baz");
    factoryBean.setPubSub(false);
    return factoryBean;
}
```

15.13.2. Configuring with the Java DSL

The following example shows how to configure the channels with the Java DSL:

```
@Bean
public IntegrationFlow pollableInFlow(ConnectionFactory connectionFactory) {
    return IntegrationFlows.from(...)
            ...
            .channel(Amqp.pollableChannel(connectionFactory)
                    .queueName("foo"))
            ...
            .get();
}

@Bean
public IntegrationFlow messageDrivenInFow(ConnectionFactory connectionFactory) {
    return IntegrationFlows.from(...)
            ...
            .channel(Amqp.channel(connectionFactory)
                    .queueName("bar"))
            ...
            .get();
}

@Bean
public IntegrationFlow pubSubInFlow(ConnectionFactory connectionFactory) {
    return IntegrationFlows.from(...)
            ...
            .channel(Amqp.publisSubscribeChannel(connectionFactory)
                    .queueName("baz"))
            ...
            .get();
}
```

15.14. AMQP Message Headers

15.14.1. Overview

The Spring Integration AMQP Adapters automatically map all AMQP properties and headers. (This is a change from 4.3 - previously, only standard headers were mapped). By default, these properties are copied to and from Spring Integration `MessageHeaders` by using the `DefaultAmqpHeaderMapper`.

You can pass in your own implementation of AMQP-specific header mappers, as the adapters have properties to support doing so.

Any user-defined headers within the AMQP `MessageProperties` are copied to or from an AMQP message, unless explicitly negated by the `requestHeaderNames` or `replyHeaderNames` properties of the `DefaultAmqpHeaderMapper`. By default, for an outbound mapper, no `x-*` headers are mapped. See the caution that appears later in this section for why.

To override the default and revert to the pre-4.3 behavior, use `STANDARD_REQUEST_HEADERS` and

STANDARD_REPLY_HEADERS in the properties.

When mapping user-defined headers, the values can also contain simple wildcard patterns (such as thing* or *thing) to be matched. The * matches all headers.

Starting with version 4.1, the AbstractHeaderMapper (a DefaultAmqpHeaderMapper superclass) lets the NON_STANDARD_HEADERS token be configured for the requestHeaderNames and replyHeaderNames properties (in addition to the existing STANDARD_REQUEST_HEADERS and STANDARD_REPLY_HEADERS) to map all user-defined headers.

The org.springframework.amqp.support.AmqpHeaders class identifies the default headers that are used by the DefaultAmqpHeaderMapper:

- amqp_appId
- amqp_clusterId
- amqp_contentEncoding
- amqp_contentLength
- content-type (see contentType Header)
- amqp_correlationId
- amqp_delay
- amqp_deliveryMode
- amqp_deliveryTag
- amqp_expiration
- amqp_messageCount
- amqp_messageId
- amqp_receivedDelay
- amqp_receivedDeliveryMode
- amqp_receivedExchange
- amqp_receivedRoutingKey
- amqp_redelivered
- amqp_replyTo
- amqp_timestamp
- amqp_type
- amqp_userId
- amqp_publishConfirm
- amqp_publishConfirmNackCause
- amqp_returnReplyCode
- amqp_returnReplyText
- amqp_returnExchange

- amqp_returnRoutingKey
- amqp_channel
- amqp_consumerTag
- amqp_consumerQueue

As mentioned earlier in this section, using a header mapping pattern of * is a common way to copy all headers. However, this can have some unexpected side effects, because certain RabbitMQ proprietary properties/headers are also copied. For example, when you use federation, the received message may have a property named x-received-from, which contains the node that sent the message. If you use the wildcard character * for the request and reply header mapping on the inbound gateway, this header is copied, which may cause some issues with federation. This reply message may be federated back to the sending broker, which may think that a message is looping and, as a result, silently drop it. If you wish to use the convenience of wildcard header mapping, you may need to filter out some headers in the downstream flow. For example, to avoid copying the x-received-from header back to the reply you can use <int:header-filter ⋯ header-names="x-received-from"> before sending the reply to the AMQP inbound gateway. Alternatively, you can explicitly list those properties that you actually want mapped, instead of using wildcards. For these reasons, for inbound messages, the mapper (by default) does not map any x-* headers. It also does not map the deliveryMode to the amqp_deliveryMode header, to avoid propagation of that header from an inbound message to an outbound message. Instead, this header is mapped to amqp_receivedDeliveryMode, which is not mapped on output.

Starting with version 4.3, patterns in the header mappings can be negated by preceding the pattern with !. Negated patterns get priority, so a list such as STANDARD_REQUEST_HEADERS,thing1,ba*,!thing2,!thing3,qux,!thing1 does not map thing1 (nor thing2 nor thing3). The standard headers plus bad and qux are mapped. The negation technique can be useful for example to not map JSON type headers for incoming messages when a JSON deserialization logic is done in the receiver downstream different way. For this purpose a !json_* pattern should be configured for header mapper of the inbound channel adapter/gateway.

If you have a user-defined header that begins with ! that you do wish to map, you need to escape it with \, as follows: STANDARD_REQUEST_HEADERS,\!myBangHeader. The header named !myBangHeader is now mapped.

Starting with version 5.1, the DefaultAmqpHeaderMapper will fall back to mapping MessageHeaders.ID and MessageHeaders.TIMESTAMP to MessageProperties.messageId and MessageProperties.timestamp respectively, if the corresponding amqp_messageId or amqp_timestamp headers are not present on outbound messages. Inbound properties will be mapped to the amqp_* headers as before. It is useful to populate the messageId property when message consumers are using stateful retry.

15.14.2. contentType Header

Unlike other headers, the `AmqpHeaders.CONTENT_TYPE` is not prefixed with `amqp_`; this allows transparent passing of the contentType header across different technologies. For example an inbound HTTP message sent to a RabbitMQ queue.

The `contentType` header is mapped to Spring AMQP's `MessageProperties.contentType` property and that is subsequently mapped to RabbitMQ's `content_type` property.

Prior to version 5.1, this header was also mapped as an entry in the `MessageProperties.headers` map; this was incorrect and, furthermore, the value could be wrong since the underlying Spring AMQP message converter might have changed the content type. Such a change would be reflected in the first-class `content_type` property, but not in the RabbitMQ headers map. Inbound mapping ignored the headers map value. `contentType` is no longer mapped to an entry in the headers map.

15.15. Strict Message Ordering

This section describes message ordering for inbound and outbound messages.

15.15.1. Inbound

If you require strict ordering of inbound messages, you must configure the inbound listener container's `prefetchCount` property to `1`. This is because, if a message fails and is redelivered, it arrives after existing prefetched messages. Since Spring AMQP version 2.0, the `prefetchCount` defaults to `250` for improved performance. Strict ordering requirements come at the cost of decreased performance.

15.15.2. Outbound

Consider the following integration flow:

```
@Bean
public IntegrationFlow flow(RabbitTemplate template) {
    return IntegrationFlows.from(Gateway.class)
            .split(s -> s.delimiters(","))
            .<String, String>transform(String::toUpperCase)
            .handle(Amqp.outboundAdapter(template).routingKey("rk"))
            .get();
}
```

Suppose we send messages `A`, `B` and `C` to the gateway. While it is likely that messages `A`, `B`, `C` are sent in order, there is no guarantee. This is because the template "borrows" a channel from the cache for each send operation, and there is no guarantee that the same channel is used for each message. One solution is to start a transaction before the splitter, but transactions are expensive in RabbitMQ and can reduce performance several hundred fold.

To solve this problem in a more efficient manner, starting with version 5.1, Spring Integration

provides the `BoundRabbitChannelAdvice` which is a `HandleMessageAdvice`. See Handling Message Advice. When applied before the splitter, it ensures that all downstream operations are performed on the same channel and, optionally, can wait until publisher confirmations for all sent messages are received (if the connection factory is configured for confirmations). The following example shows how to use `BoundRabbitChannelAdvice`:

```
@Bean
public IntegrationFlow flow(RabbitTemplate template) {
    return IntegrationFlows.from(Gateway.class)
            .split(s -> s.delimiters(",")
                    .advice(new BoundRabbitChannelAdvice(template, Duration
.ofSeconds(10))))
            .<String, String>transform(String::toUpperCase)
            .handle(Amqp.outboundAdapter(template).routingKey("rk"))
            .get();
}
```

Notice that the same `RabbitTemplate` (which implements `RabbitOperations`) is used in the advice and the outbound adapter. The advice runs the downstream flow within the template's `invoke` method so that all operations run on the same channel. If the optional timeout is provided, when the flow completes, the advice calls the `waitForConfirmsOrDie` method, which throws an exception if the confirmations are not received within the specified time.

There must be no thread handoffs in the downstream flow (`QueueChannel`, `ExecutorChannel`, and others).

15.16. AMQP Samples

To experiment with the AMQP adapters, check out the samples available in the Spring Integration samples git repository at https://github.com/SpringSource/spring-integration-samples

Currently, one sample demonstrates the basic functionality of the Spring Integration AMQP adapter by using an outbound channel adapter and an inbound channel adapter. As AMQP broker implementation in the sample uses RabbitMQ.

In order to run the example, you need a running instance of RabbitMQ. A local installation with just the basic defaults suffices. For detailed RabbitMQ installation procedures, see https://www.rabbitmq.com/install.html

Once the sample application is started, enter some text on the command prompt and a message containing that entered text is dispatched to the AMQP queue. In return, that message is retrieved by Spring Integration and printed to the console.

The following image illustrates the basic set of Spring Integration components used in this sample.

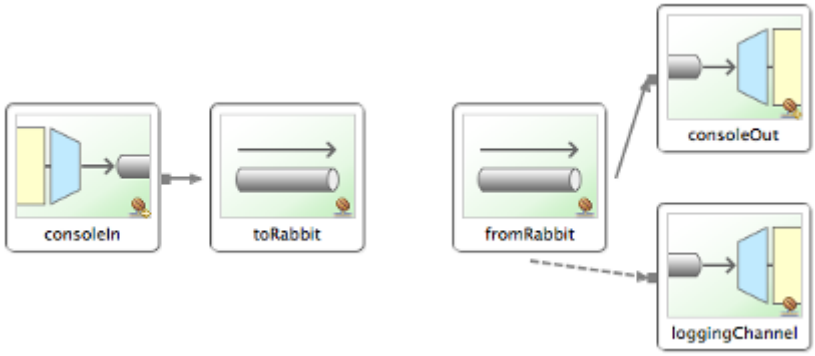


Figure 7. The Spring Integration graph of the AMQP sample

Chapter 16. Spring `ApplicationEvent` Support

Spring Integration provides support for inbound and outbound `ApplicationEvents`, as defined by the underlying Spring Framework. For more information about Spring's support for events and listeners, see the Spring Reference Manual.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-event</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-event:5.3.0.RC1"
```

16.1. Receiving Spring Application Events

To receive events and send them to a channel, you can define an instance of Spring Integration's `ApplicationEventListeningMessageProducer`. This class is an implementation of Spring's `ApplicationListener` interface. By default, it passes all received events as Spring Integration messages. To limit based on the type of event, you can use the 'eventTypes' property to configure the list of event types that you want to receive. If a received event has a `Message` instance as its 'source', that `Message` is passed as-is. Otherwise, if a SpEL-based `payloadExpression` has been provided, that is evaluated against the `ApplicationEvent` instance. If the event's source is not a `Message` instance and no `payloadExpression` has been provided, the `ApplicationEvent` itself is passed as the payload.

Starting with version 4.2, the `ApplicationEventListeningMessageProducer` implements `GenericApplicationListener` and can be configured to accept not only `ApplicationEvent` types but any type for treating payload events (which are also supported since Spring Framework 4.2). When the accepted event is an instance of `PayloadApplicationEvent`, its `payload` is used for the message to send.

For convenience, namespace support is provided to configure an `ApplicationEventListeningMessageProducer` with the `inbound-channel-adapter` element, as the following example shows:

```
<int-event:inbound-channel-adapter channel="eventChannel"
                                   error-channel="eventErrorChannel"
                                   event-types="example.FooEvent,
example.BarEvent, java.util.Date"/>

<int:publish-subscribe-channel id="eventChannel"/>
```

In the preceding example, all application context events that match one of the types specified by the 'event-types' (optional) attribute are delivered as Spring Integration messages to the message channel named 'eventChannel'. If a downstream component throws an exception, a `MessagingException` that contains the failed message and exception is sent to the channel named 'eventErrorChannel'. If no `error-channel` is specified and the downstream channels are synchronous, the exception is propagated to the caller.

Using Java to configure the same adapter:

```
@Bean
public ApplicationEventListeningMessageProducer eventsAdapter(
            MessageChannel eventChannel, MessageChannel eventErrorChannel) {

    ApplicationEventListeningMessageProducer producer =
        new ApplicationEventListeningMessageProducer();
    producer.setEventTypes(example.FooEvent.class, example.BarEvent.class, java
.util.Date.class);
    producer.setOutputChannel(eventChannel);
    producer.setErrorChannel(eventErrorChannel);
    return producer;
}
```

With the Java DSL:

```
@Bean
public ApplicationEventListeningMessageProducer eventsAdapter() {

    ApplicationEventListeningMessageProducer producer =
        new ApplicationEventListeningMessageProducer();
    producer.setEventTypes(example.FooEvent.class, example.BarEvent.class, java
.util.Date.class);
    return producer;
}

@Bean
public IntegrationFlow eventFlow(ApplicationEventListeningMessageProducer
eventsAdapter,
        MessageChannel eventErrorChannel) {

    return IntegrationFlows.from(eventsAdapter, e -> e.errorChannel
(eventErrorChannel))
        .handle(...)
        ...
        .get();
}
```

16.2. Sending Spring Application Events

To send Spring `ApplicationEvents`, create an instance of the `ApplicationEventPublishingMessageHandler` and register it within an endpoint. This implementation of the `MessageHandler` interface also implements Spring's `ApplicationEventPublisherAware` interface and consequently acts as a bridge between Spring Integration messages and `ApplicationEvents`.

For convenience, namespace support is provided to configure an `ApplicationEventPublishingMessageHandler` with the `outbound-channel-adapter` element, as the following example shows:

```
<int:channel id="eventChannel"/>

<int-event:outbound-channel-adapter channel="eventChannel"/>
```

If you use a `PollableChannel` (such as a `QueueChannel`), you can also provide a `poller` child element of the `outbound-channel-adapter` element. You can also optionally provide a `task-executor` reference for that poller. The following example demonstrates both:

```
<int:channel id="eventChannel">
  <int:queue/>
</int:channel>

<int-event:outbound-channel-adapter channel="eventChannel">
  <int:poller max-messages-per-poll="1" task-executor="executor" fixed-rate="100
"/>
</int-event:outbound-channel-adapter>

<task:executor id="executor" pool-size="5"/>
```

In the preceding example, all messages sent to the 'eventChannel' channel are published as `ApplicationEvent` instances to any relevant `ApplicationListener` instances that are registered within the same Spring `ApplicationContext`. If the payload of the message is an `ApplicationEvent`, it is passed as-is. Otherwise, the message itself is wrapped in a `MessagingEvent` instance.

Starting with version 4.2, you can configure the `ApplicationEventPublishingMessageHandler` (`<int-event:outbound-channel-adapter>`) with the `publish-payload` boolean attribute to publish to the application context `payload` as is, instead of wrapping it to a `MessagingEvent` instance.

To configure the adapter using Java configuration:

```
@Bean
@ServiceActivator(inputChannel = "eventChannel")
public ApplicationEventPublishingMessageHandler eventHandler() {
    ApplicationEventPublishingMessageHandler handler =
            new ApplicationEventPublishingMessageHandler();
    handler.setPublishPayload(true);
    return handler;
}
```

With the Java DSL:

```
@Bean
public ApplicationEventPublishingMessageHandler eventHandler() {
    ApplicationEventPublishingMessageHandler handler =
            new ApplicationEventPublishingMessageHandler();
    handler.setPublishPayload(true);
    return handler;
}

@Bean
// MessageChannel is "eventsFlow.input"
public IntegrationFlow eventsOutFlow(ApplicationEventPublishingMessageHandler
eventHandler) {
    return f -> f.handle(eventHandler);
}
```

Chapter 17. Feed Adapter

Spring Integration provides support for syndication through feed adapters. The implementation is based on the ROME Framework.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-feed</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-feed:5.3.0.RC1"
```

Web syndication is a way to publish material such as news stories, press releases, blog posts, and other items typically available on a website but also made available in a feed format such as RSS or ATOM.

Spring integration provides support for web syndication through its 'feed' adapter and provides convenient namespace-based configuration for it. To configure the 'feed' namespace, include the following elements within the headers of your XML configuration file:

```
xmlns:int-feed="http://www.springframework.org/schema/integration/feed"
xsi:schemaLocation="http://www.springframework.org/schema/integration/feed
    https://www.springframework.org/schema/integration/feed/spring-integration-
feed.xsd"
```

17.1. Feed Inbound Channel Adapter

The only adapter you really need to provide support for retrieving feeds is an inbound channel adapter. It lets you subscribe to a particular URL. The following example shows a possible configuration:

```
<int-feed:inbound-channel-adapter id="feedAdapter"
        channel="feedChannel"
        url="https://feeds.bbci.co.uk/news/rss.xml">
    <int:poller fixed-rate="10000" max-messages-per-poll="100" />
</int-feed:inbound-channel-adapter>
```

In the preceding configuration, we are subscribing to a URL identified by the `url` attribute.

As news items are retrieved, they are converted to messages and sent to a channel identified by the `channel` attribute. The payload of each message is a `com.sun.syndication.feed.synd.SyndEntry` instance. Each one encapsulates various data about a news item (content, dates, authors, and other details).

The inbound feed channel adapter is a polling consumer. That means that you must provide a poller configuration. However, one important thing you must understand with regard to a feed is that its inner workings are slightly different then most other polling consumers. When an inbound feed adapter is started, it does the first poll and receives a `com.sun.syndication.feed.synd.SyndEntryFeed` instance. That object contains multiple `SyndEntry` objects. Each entry is stored in the local entry queue and is released based on the value in the `max-messages-per-poll` attribute, such that each message contains a single entry. If, during retrieval of the entries from the entry queue, the queue has become empty, the adapter attempts to update the feed, thereby populating the queue with more entries (`SyndEntry` instances), if any are available. Otherwise the next attempt to poll for a feed is determined by the trigger of the poller (every ten seconds in the preceding configuration).

17.2. Duplicate Entries

Polling for a feed can result in entries that have already been processed ("I already read that news item, why are you showing it to me again?"). Spring Integration provides a convenient mechanism to eliminate the need to worry about duplicate entries. Each feed entry has a "published date" field. Every time a new `Message` is generated and sent, Spring Integration stores the value of the latest published date in an instance of the `MetadataStore` strategy (see Metadata Store).

The key used to persist the latest published date is the value of the (required) `id` attribute of the feed inbound channel adapter component plus the `feedUrl` (if any) from the adapter's configuration.

17.3. Other Options

Starting with version 5.0, the deprecated `com.rometools.fetcher.FeedFetcher` option has been removed and an overloaded `FeedEntryMessageSource` constructor for an `org.springframework.core.io.Resource` is provided. This is useful when the feed source is not an HTTP endpoint but is any other resource (such as local or remote on FTP). In the `FeedEntryMessageSource` logic, such a resource (or provided `URL`) is parsed by the `SyndFeedInput` to the `SyndFeed` object for the processing mentioned earlier. You can also inject a customized `SyndFeedInput`

(for example, with the `allowDoctypes` option) instance into the `FeedEntryMessageSource`.

17.4. Java DSL Configuration

The following Spring Boot application shows an example of how to configure the inbound adapter with the Java DSL:

```
@SpringBootApplication
public class FeedJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FeedJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Value("org/springframework/integration/feed/sample.rss")
    private Resource feedResource;

    @Bean
    public MetadataStore metadataStore() {
        PropertiesPersistingMetadataStore metadataStore = new
PropertiesPersistingMetadataStore();
        metadataStore.setBaseDirectory(tempFolder.getRoot().getAbsolutePath());
        return metadataStore;
    }

    @Bean
    public IntegrationFlow feedFlow() {
        return IntegrationFlows
                .from(Feed.inboundAdapter(this.feedResource, "feedTest")
                                .metadataStore(metadataStore()),
                        e -> e.poller(p -> p.fixedDelay(100)))
                .channel(c -> c.queue("entries"))
                .get();
    }

}
```

Chapter 18. File Support

Spring Integration's file support extends the Spring Integration core with a dedicated vocabulary to deal with reading, writing, and transforming files.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-file</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-file:5.3.0.RC1"
```

It provides a namespace that enables elements defining channel adapters dedicated to files and support for transformers that can read file contents into strings or byte arrays.

This section explains the workings of `FileReadingMessageSource` and `FileWritingMessageHandler` and how to configure them as beans. It also discusses the support for dealing with files through file-specific implementations of `Transformer`. Finally, it explains the file-specific namespace.

18.1. Reading Files

A `FileReadingMessageSource` can be used to consume files from the filesystem. This is an implementation of `MessageSource` that creates messages from a file system directory. The following example shows how to configure a `FileReadingMessageSource`:

```
<bean id="pollableFileSource"
    class="org.springframework.integration.file.FileReadingMessageSource"
    p:directory="${input.directory}"/>
```

To prevent creating messages for certain files, you can supply a `FileListFilter`. By default, we use the following filters:

- `IgnoreHiddenFileListFilter`
- `AcceptOnceFileListFilter`

The `IgnoreHiddenFileListFilter` ensures that hidden files are not processed. Note that the exact

definition of hidden is system-dependent. For example, on UNIX-based systems, a file beginning with a period character is considered to be hidden. Microsoft Windows, on the other hand, has a dedicated file attribute to indicate hidden files.

Version 4.2 introduced the `IgnoreHiddenFileListFilter`. In prior versions, hidden files were included. With the default configuration, the `IgnoreHiddenFileListFilter` is triggered first, followed by the `AcceptOnceFileListFilter`.

The `AcceptOnceFileListFilter` ensures files are picked up only once from the directory.

The `AcceptOnceFileListFilter` stores its state in memory. If you wish the state to survive a system restart, you can use the `FileSystemPersistentAcceptOnceFileListFilter`. This filter stores the accepted file names in a `MetadataStore` implementation (see Metadata Store). This filter matches on the filename and modified time.

Since version 4.0, this filter requires a `ConcurrentMetadataStore`. When used with a shared data store (such as `Redis` with the `RedisMetadataStore`), it lets filter keys be shared across multiple application instances or across a network file share being used by multiple servers.

Since version 4.1.5, this filter has a new property (`flushOnUpdate`), which causes it to flush the metadata store on every update (if the store implements `Flushable`).

The following example configures a `FileReadingMessageSource` with a filter:

```
<bean id="pollableFileSource"
    class="org.springframework.integration.file.FileReadingMessageSource"
    p:inputDirectory="${input.directory}"
    p:filter-ref="customFilterBean"/>
```

A common problem with reading files is that a file may be detected before it is ready (that is, some other process may still be writing the file). The default `AcceptOnceFileListFilter` does not prevent this. In most cases, this can be prevented if the file-writing process renames each file as soon as it is ready for reading. A `filename-pattern` or `filename-regex` filter that accepts only files that are ready (perhaps based on a known suffix), composed with the default `AcceptOnceFileListFilter`, allows for this situation. The `CompositeFileListFilter` enables the composition, as the following example shows:

```
<bean id="pollableFileSource"
    class="org.springframework.integration.file.FileReadingMessageSource"
    p:inputDirectory="${input.directory}"
    p:filter-ref="compositeFilter"/>

<bean id="compositeFilter"
    class="org.springframework.integration.file.filters.CompositeFileListFilter">
    <constructor-arg>
        <list>
            <bean class="o.s.i.file.filters.AcceptOnceFileListFilter"/>
            <bean class="o.s.i.file.filters.RegexPatternFileListFilter">
                <constructor-arg value="^test.*$"/>
            </bean>
        </list>
    </constructor-arg>
</bean>
```

If it is not possible to create the file with a temporary name and rename to the final name, Spring Integration provides another alternative. Version 4.2 added the `LastModifiedFileListFilter`. This filter can be configured with an `age` property so that only files older than this value are passed by the filter. The age defaults to 60 seconds, but you should choose an age that is large enough to avoid picking up a file early (due to, say, network glitches). The following example shows how to configure a `LastModifiedFileListFilter`:

```
<bean id="filter" class=
"org.springframework.integration.file.filters.LastModifiedFileListFilter">
    <property name="age" value="120" />
</bean>
```

Starting with version 4.3.7, a `ChainFileListFilter` (an extension of `CompositeFileListFilter`) has been introduced to allow scenarios when subsequent filters should only see the result of the previous filter. (With the `CompositeFileListFilter`, all filters see all the files, but it passes only files that have passed all filters). An example of where the new behavior is required is a combination of `LastModifiedFileListFilter` and `AcceptOnceFileListFilter`, when we do not wish to accept the file until some amount of time has elapsed. With the `CompositeFileListFilter`, since the `AcceptOnceFileListFilter` sees all the files on the first pass, it does not pass it later when the other filter does. The `CompositeFileListFilter` approach is useful when a pattern filter is combined with a custom filter that looks for a secondary file to indicate that file transfer is complete. The pattern filter might only pass the primary file (such as `something.txt`) but the “done” filter needs to see whether (for example) `something.done` is present.

Say we have files `a.txt`, `a.done`, and `b.txt`.

The pattern filter passes only `a.txt` and `b.txt`, while the “done” filter sees all three files and passes

only `a.txt`. The final result of the composite filter is that only `a.txt` is released.

With the `ChainFileListFilter`, if any filter in the chain returns an empty list, the remaining filters are not invoked.

Version 5.0 introduced an `ExpressionFileListFilter` to execute SpEL expression against a file as a context evaluation root object. For this purpose, all the XML components for file handling (local and remote), along with an existing `filter` attribute, have been supplied with the `filter-expression` option, as the following example shows:

```
<int-file:inbound-channel-adapter
        directory="${inputdir}"
        filter-expression="name matches '.text'"
        auto-startup="false"/>
```

Version 5.0.5 introduced the `DiscardAwareFileListFilter` implementations that have an interest in rejected files. For this purpose, such a filter implementation should be supplied with a callback through `addDiscardCallback(Consumer<File>)`. In the framework, this functionality is used from the `FileReadingMessageSource.WatchServiceDirectoryScanner`, in combination with `LastModifiedFileListFilter`. Unlike the regular `DirectoryScanner`, the `WatchService` provides files for processing according to the events on the target file system. At the moment of polling an internal queue with those files, the `LastModifiedFileListFilter` may discard them because they are too young relative to its configured `age`. Therefore, we lose the file for future possible considerations. The discard callback hook lets us retain the file in the internal queue so that it is available to be checked against the `age` in subsequent polls. The `CompositeFileListFilter` also implements a `DiscardAwareFileListFilter` and populates a discard callback to all its `DiscardAwareFileListFilter` delegates.

Since `CompositeFileListFilter` matches the files against all delegates, the `discardCallback` may be called several times for the same file.

Starting with version 5.1, the `FileReadingMessageSource` doesn't check a directory for existence and doesn't create it until its `start()` is called (typically via wrapping `SourcePollingChannelAdapter`). Previously, there was no simple way to prevent an operation system permissions error when referencing the directory, for example from tests, or when permissions are applied later.

18.1.1. Message Headers

Starting with version 5.0, the `FileReadingMessageSource` (in addition to the `payload` as a polled `File`) populates the following headers to the outbound `Message`:

- `FileHeaders.FILENAME`: The `File.getName()` of the file to send. Can be used for subsequent rename or copy logic.
- `FileHeaders.ORIGINAL_FILE`: The `File` object itself. Typically, this header is populated automatically by framework components (such as splitters or transformers) when we lose the

original `File` object. However, for consistency and convenience with any other custom use cases, this header can be useful to get access to the original file.

- `FileHeaders.RELATIVE_PATH`: A new header introduced to represent the part of file path relative to the root directory for the scan. This header can be useful when the requirement is to restore a source directory hierarchy in the other places. For this purpose, the `DefaultFileNameGenerator` (see "`Generating File Names) can be configured to use this header.

18.1.2. Directory Scanning and Polling

The `FileReadingMessageSource` does not produce messages for files from the directory immediately. It uses an internal queue for 'eligible files' returned by the `scanner`. The `scanEachPoll` option is used to ensure that the internal queue is refreshed with the latest input directory content on each poll. By default (`scanEachPoll = false`), the `FileReadingMessageSource` empties its queue before scanning the directory again. This default behavior is particularly useful to reduce scans of large numbers of files in a directory. However, in cases where custom ordering is required, it is important to consider the effects of setting this flag to `true`. The order in which files are processed may not be as expected. By default, files in the queue are processed in their natural (`path`) order. New files added by a scan, even when the queue already has files, are inserted in the appropriate position to maintain that natural order. To customize the order, the `FileReadingMessageSource` can accept a `Comparator<File>` as a constructor argument. It is used by the internal (`PriorityBlockingQueue`) to reorder its content according to the business requirements. Therefore, to process files in a specific order, you should provide a comparator to the `FileReadingMessageSource` rather than ordering the list produced by a custom `DirectoryScanner`.

Version 5.0 introduced `RecursiveDirectoryScanner` to perform file tree visiting. The implementation is based on the `Files.walk(Path start, int maxDepth, FileVisitOption··· options)` functionality. The root directory (`DirectoryScanner.listFiles(File)`) argument is excluded from the result. All other sub-directories inclusions and exclusions are based on the target `FileListFilter` implementation. For example, the `SimplePatternFileListFilter` filters out directories by default. See `AbstractDirectoryAwareFileListFilter` and its implementations for more information.

18.1.3. Namespace Support

The configuration for file reading can be simplified by using the file-specific namespace. To do so, use the following template:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-file="http://www.springframework.org/schema/integration/file"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    https://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/file
    https://www.springframework.org/schema/integration/file/spring-integration-
file.xsd">
</beans>
```

Within this namespace, you can reduce the `FileReadingMessageSource` and wrap it in an inbound Channel Adapter, as follows:

```
<int-file:inbound-channel-adapter id="filesIn1"
    directory="file:${input.directory}" prevent-duplicates="true" ignore-hidden=
"true"/>

<int-file:inbound-channel-adapter id="filesIn2"
    directory="file:${input.directory}"
    filter="customFilterBean" />

<int-file:inbound-channel-adapter id="filesIn3"
    directory="file:${input.directory}"
    filename-pattern="test*" />

<int-file:inbound-channel-adapter id="filesIn4"
    directory="file:${input.directory}"
    filename-regex="test[0-9]+\.txt" />
```

The first channel adapter example relies on the default `FileListFilter` implementations:

- `IgnoreHiddenFileListFilter` (do not process hidden files)
- `AcceptOnceFileListFilter` (prevent duplication)

Therefore, you can also leave off the `prevent-duplicates` and `ignore-hidden` attributes, as they are `true` by default.

Spring Integration 4.2 introduced the `ignore-hidden` attribute. In prior versions, hidden files were included.

The second channel adapter example uses a custom filter, the third uses the `filename-pattern` attribute to add an `AntPathMatcher` based filter, and the fourth uses the `filename-regex` attribute to add a regular expression pattern-based filter to the `FileReadingMessageSource`. The `filename-pattern` and `filename-regex` attributes are each mutually exclusive with the regular `filter` reference attribute. However, you can use the `filter` attribute to reference an instance of `CompositeFileListFilter` that combines any number of filters, including one or more pattern-based filters to fit your particular needs.

When multiple processes read from the same directory, you may want to lock files to prevent them from being picked up concurrently. To do so, you can use a `FileLocker`. There is a `java.nio`-based implementation available, but it is also possible to implement your own locking scheme. The `nio` locker can be injected as follows:

```
<int-file:inbound-channel-adapter id="filesIn"
    directory="file:${input.directory}" prevent-duplicates="true">
    <int-file:nio-locker/>
</int-file:inbound-channel-adapter>
```

You can configure a custom locker as follows:

```
<int-file:inbound-channel-adapter id="filesIn"
    directory="file:${input.directory}" prevent-duplicates="true">
    <int-file:locker ref="customLocker"/>
</int-file:inbound-channel-adapter>
```

When a file inbound adapter is configured with a locker, it takes responsibility for acquiring a lock before the file is allowed to be received. It does not assume the responsibility to unlock the file. If you have processed the file and keep the locks hanging around, you have a memory leak. If this is a problem, you should call `FileLocker.unlock(File file)` yourself at the appropriate time.

When filtering and locking files is not enough, you might need to control the way files are listed entirely. To implement this type of requirement, you can use an implementation of `DirectoryScanner`. This scanner lets you determine exactly what files are listed in each poll. This is also the interface that Spring Integration uses internally to wire `FileListFilter` instances and `FileLocker` to the `FileReadingMessageSource`. You can inject a custom `DirectoryScanner` into the `<int-file:inbound-channel-adapter/>` on the `scanner` attribute, as the following example shows:

```
<int-file:inbound-channel-adapter id="filesIn" directory="file:${input.directory}"
     scanner="customDirectoryScanner"/>
```

Doing so gives you full freedom to choose the ordering, listing, and locking strategies.

It is also important to understand that filters (including `patterns`, `regex`, `prevent-duplicates`, and others) and `locker` instances are actually used by the `scanner`. Any of these attributes set on the adapter are subsequently injected into the internal `scanner`. For the case of an external `scanner`, all filter and locker attributes are prohibited on the `FileReadingMessageSource`. They must be specified (if required) on that custom `DirectoryScanner`. In other words, if you inject a `scanner` into the `FileReadingMessageSource`, you should supply `filter` and `locker` on that `scanner`, not on the `FileReadingMessageSource`.

By default, the `DefaultDirectoryScanner` uses an `IgnoreHiddenFileListFilter` and an `AcceptOnceFileListFilter`. To prevent their use, you can configure your own filter (such as `AcceptAllFileListFilter`) or even set it to `null`.

18.1.4. `WatchServiceDirectoryScanner`

The `FileReadingMessageSource.WatchServiceDirectoryScanner` relies on file-system events when new files are added to the directory. During initialization, the directory is registered to generate events. The initial file list is also built during initialization. While walking the directory tree, any subdirectories encountered are also registered to generate events. On the first poll, the initial file list from walking the directory is returned. On subsequent polls, files from new creation events are returned. If a new subdirectory is added, its creation event is used to walk the new subtree to find existing files and register any new subdirectories found.

There is an issue with `WatchKey` when its internal events `queue` is not drained by the program as quickly as the directory modification events occur. If the queue size is exceeded, a `StandardWatchEventKinds.OVERFLOW` is emitted to indicate that some file system events may be lost. In this case, the root directory is re-scanned completely. To avoid duplicates, consider using an appropriate `FileListFilter` (such as the `AcceptOnceFileListFilter`) or removing files when processing is complete.

The `WatchServiceDirectoryScanner` can be enabled through the `FileReadingMessageSource.use-watch-service` option, which is mutually exclusive with the `scanner` option. An internal `FileReadingMessageSource.WatchServiceDirectoryScanner` instance is populated for the provided `directory`.

In addition, now the `WatchService` polling logic can track the `StandardWatchEventKinds.ENTRY_MODIFY` and `StandardWatchEventKinds.ENTRY_DELETE`.

If you need to track the modification of existing files as well as new files, you should implement the `ENTRY_MODIFY` events logic in the `FileListFilter`. Otherwise, the files from those events are treated the same way.

The `ResettableFileListFilter` implementations pick up the `ENTRY_DELETE` events. Consequently, their files are provided for the `remove()` operation. When this event is enabled, filters such as the `AcceptOnceFileListFilter` have the file removed As a result, if a file with the same name appears, it passes the filter and is sent as a message.

For this purpose, the `watch-events` property

(`FileReadingMessageSource.setWatchEvents(WatchEventType… watchEvents)`) has been introduced. (`WatchEventType` is a public inner enumeration in `FileReadingMessageSource`.) With such an option, we can use one downstream flow logic for new files and use some other logic for modified files. The following example shows how to configure different logic for create and modify events in the same directory:

```
<int-file:inbound-channel-adapter id="newFiles"
     directory="${input.directory}"
     use-watch-service="true"/>

<int-file:inbound-channel-adapter id="modifiedFiles"
     directory="${input.directory}"
     use-watch-service="true"
     filter="acceptAllFilter"
     watch-events="MODIFY"/> <!-- The default is CREATE. -->
```

18.1.5. Limiting Memory Consumption

You can use a `HeadDirectoryScanner` to limit the number of files retained in memory. This can be useful when scanning large directories. With XML configuration, this is enabled by setting the `queue-size` property on the inbound channel adapter.

Prior to version 4.2, this setting was incompatible with the use of any other filters. Any other filters (including `prevent-duplicates="true"`) overwrote the filter used to limit the size.

The use of a `HeadDirectoryScanner` is incompatible with an `AcceptOnceFileListFilter`. Since all filters are consulted during the poll decision, the `AcceptOnceFileListFilter` does not know that other filters might be temporarily filtering files. Even if files that were previously filtered by the `HeadDirectoryScanner.HeadFilter` are now available, the `AcceptOnceFileListFilter` filters them.

Generally, instead of using an `AcceptOnceFileListFilter` in this case, you should remove the processed files so that the previously filtered files are available on a future poll.

18.1.6. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the outbound adapter with Java configuration:

```
@SpringBootApplication
public class FileReadingJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FileReadingJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public MessageChannel fileInputChannel() {
        return new DirectChannel();
    }

    @Bean
    @InboundChannelAdapter(value = "fileInputChannel", poller = @Poller(fixedDelay
= "1000"))
    public MessageSource<File> fileReadingMessageSource() {
         FileReadingMessageSource source = new FileReadingMessageSource();
         source.setDirectory(new File(INBOUND_PATH));
         source.setFilter(new SimplePatternFileListFilter("*.txt"));
         return source;
    }

    @Bean
    @Transformer(inputChannel = "fileInputChannel", outputChannel =
"processFileChannel")
    public FileToStringTransformer fileToStringTransformer() {
        return new FileToStringTransformer();
    }

}
```

18.1.7. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the outbound adapter with the Java DSL:

```
@SpringBootApplication
public class FileReadingJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FileReadingJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public IntegrationFlow fileReadingFlow() {
         return IntegrationFlows
                  .from(Files.inboundAdapter(new File(INBOUND_PATH))
                              .patternFilter("*.txt"),
                          e -> e.poller(Pollers.fixedDelay(1000)))
                  .transform(Files.toStringTransformer())
                  .channel("processFileChannel")
                  .get();
    }

}
```

18.1.8. 'tail'ing Files

Another popular use case is to get 'lines' from the end (or tail) of a file, capturing new lines when they are added. Two implementations are provided. The first, `OSDelegatingFileTailingMessageProducer`, uses the native `tail` command (on operating systems that have one). This is generally the most efficient implementation on those platforms. For operating systems that do not have a `tail` command, the second implementation, `ApacheCommonsFileTailingMessageProducer`, uses the Apache `commons-io Tailer` class.

In both cases, file system events, such as files being unavailable and other events, are published as `ApplicationEvent` instances by using the normal Spring event publishing mechanism. Examples of such events include the following:

```
[message=tail: cannot open '/tmp/somefile' for reading:
               No such file or directory, file=/tmp/somefile]

[message=tail: '/tmp/somefile' has become accessible, file=/tmp/somefile]

[message=tail: '/tmp/somefile' has become inaccessible:
               No such file or directory, file=/tmp/somefile]

[message=tail: '/tmp/somefile' has appeared;
               following end of new file, file=/tmp/somefile]
```

The sequence of events shown in the preceding example might occur, for example, when a file is rotated.

Starting with version 5.0, a `FileTailingIdleEvent` is emitted when there is no data in the file during `idleEventInterval`. The following example shows what such an event looks like:

```
[message=Idle timeout, file=/tmp/somefile] [idle time=5438]
```

Not all platforms that support a `tail` command provide these status messages.

Messages emitted from these endpoints have the following headers:

- `FileHeaders.ORIGINAL_FILE`: The `File` object
- `FileHeaders.FILENAME`: The file name (`File.getName()`)

In versions prior to version 5.0, the `FileHeaders.FILENAME` header contained a string representation of the file's absolute path. You can now obtain that string representation by calling `getAbsolutePath()` on the original file header.

The following example creates a native adapter with the default options ('-F -n 0', meaning to follow the file name from the current end).

```
<int-file:tail-inbound-channel-adapter id="native"
    channel="input"
    task-executor="exec"
    file="/tmp/foo"/>
```

The following example creates a native adapter with '-F -n +0' options (meaning follow the file name, emitting all existing lines).

```
<int-file:tail-inbound-channel-adapter id="native"
    channel="input"
    native-options="-F -n +0"
    task-executor="exec"
    file-delay=10000
    file="/tmp/foo"/>
```

If the `tail` command fails (on some platforms, a missing file causes the `tail` to fail, even with `-F` specified), the command is retried every 10 seconds.

By default, native adapters capture from standard output and send the content as messages. They also capture from standard error to raise events. Starting with version 4.3.6, you can discard the standard error events by setting the `enable-status-reader` to `false`, as the following example shows:

```
<int-file:tail-inbound-channel-adapter id="native"
    channel="input"
    enable-status-reader="false"
    task-executor="exec"
    file="/tmp/foo"/>
```

In the following example, `IdleEventInterval` is set to `5000`, meaning that, if no lines are written for five seconds, `FileTailingIdleEvent` is triggered every five seconds:

```
<int-file:tail-inbound-channel-adapter id="native"
    channel="input"
    idle-event-interval="5000"
    task-executor="exec"
    file="/tmp/somefile"/>
```

This can be useful when you need to stop the adapter.

The following example creates an Apache `commons-io Tailer` adapter that examines the file for new lines every two seconds and checks for existence of a missing file every ten seconds:

```
<int-file:tail-inbound-channel-adapter id="apache"
    channel="input"
    task-executor="exec"
    file="/tmp/bar"
    delay="2000"
    end="false"             ①
    reopen="true"           ②
    file-delay="10000"/>
```

① The file is tailed from the beginning (`end="false"`) instead of the end (which is the default).

② The file is reopened for each chunk (the default is to keep the file open).

Specifying the `delay`, `end` or `reopen` attributes forces the use of the Apache `commons-io` adapter and makes the `native-options` attribute unavailable.

18.1.9. Dealing With Incomplete Data

A common problem in file-transfer scenarios is how to determine that the transfer is complete so that you do not start reading an incomplete file. A common technique to solve this problem is to write the file with a temporary name and then atomically rename it to the final name. This technique, together with a filter that masks the temporary file from being picked up by the consumer, provides a robust solution. This technique is used by Spring Integration components that write files (locally or remotely). By default, they append `.writing` to the file name and remove it when the transfer is complete.

Another common technique is to write a second "marker" file to indicate that the file transfer is complete. In this scenario, you should not consider `somefile.txt` (for example) to be available for use until `somefile.txt.complete` is also present. Spring Integration version 5.0 introduced new filters to support this mechanism. Implementations are provided for the file system (`FileSystemMarkerFilePresentFileListFilter`), FTP and SFTP. They are configurable such that the marker file can have any name, although it is usually related to the file being transferred. See the Javadoc for more information.

18.2. Writing files

To write messages to the file system, you can use a `FileWritingMessageHandler`. This class can deal with the following payload types:

- `File`
- `String`
- byte array
- `InputStream` (since *version 4.2*)

For a String payload, you can configure the encoding and the charset.

To make things easier, you can configure the `FileWritingMessageHandler` as part of an outbound channel adapter or outbound gateway by using the XML namespace.

Starting with version 4.3, you can specify the buffer size to use when writing files.

Starting with version 5.1, you can provide a `BiConsumer<File, Message<?>> newFileCallback` which is triggered if you use `FileExistsMode.APPEND` or `FileExistsMode.APPEND_NO_FLUSH` and a new file has to be created. This callback receives a newly created file and the message which triggered it. This callback could be used to write a CSV header defined in the message header, for an example.

18.2.1. Generating File Names

In its simplest form, the `FileWritingMessageHandler` requires only a destination directory for writing the files. The name of the file to be written is determined by the handler's `FileNameGenerator`. The default implementation looks for a message header whose key matches the constant defined as `FileHeaders.FILENAME`.

Alternatively, you can specify an expression to be evaluated against the message to generate a file name — for example, `headers['myCustomHeader'] + '.something'`. The expression must evaluate to a `String`. For convenience, the `DefaultFileNameGenerator` also provides the `setHeaderName` method, letting you explicitly specify the message header whose value is to be used as the filename.

Once set up, the `DefaultFileNameGenerator` employs the following resolution steps to determine the filename for a given message payload:

1. Evaluate the expression against the message and, if the result is a non-empty `String`, use it as the filename.
2. Otherwise, if the payload is a `java.io.File`, use the `File` object's filename.
3. Otherwise, use the message ID appended with `.msg` as the filename.

When you use the XML namespace support, both the file outbound channel adapter and the file outbound gateway support the following mutually exclusive configuration attributes:

- `filename-generator` (a reference to a `FileNameGenerator` implementation)
- `filename-generator-expression` (an expression that evaluates to a `String`)

While writing files, a temporary file suffix is used (its default is `.writing`). It is appended to the filename while the file is being written. To customize the suffix, you can set the `temporary-file-suffix` attribute on both the file outbound channel adapter and the file outbound gateway.

When using the `APPEND` file `mode`, the `temporary-file-suffix` attribute is ignored, since the data is appended to the file directly.

Starting with ,version 4.2.5, the generated file name (as a result of `filename-generator` or `filename-generator-expression` evaluation) can represent a child path together with the target file name. It is used as a second constructor argument for `File(File parent, String child)` as before. However, in the past we did not create (`mkdirs()`) directories for the child path, assuming only the file name. This approach is useful for cases when we need to restore the file system tree to match the source

directory — for example, when unzipping the archive and saving all the files in the target directory in the original order.

18.2.2. Specifying the Output Directory

Both, the file outbound channel adapter and the file outbound gateway provide two mutually exclusive configuration attributes for specifying the output directory:

- `directory`
- `directory-expression`

Spring Integration 2.2 introduced the `directory-expression` attribute.

Using the `directory` Attribute

When you use the `directory` attribute, the output directory is set to a fixed value, which is set when the `FileWritingMessageHandler` is initialized. If you do not specify this attribute, you must use the `directory-expression` attribute.

Using the `directory-expression` Attribute

If you want to have full SpEL support, you can use the `directory-expression` attribute. This attribute accepts a SpEL expression that is evaluated for each message being processed. Thus, you have full access to a message's payload and its headers when you dynamically specify the output file directory.

The SpEL expression must resolve to either a `String`, `java.io.File` or `org.springframework.core.io.Resource`. (The later is evaluated into a `File` anyway.) Furthermore, the resulting `String` or `File` must point to a directory. If you do not specify the `directory-expression` attribute, then you must set the `directory` attribute.

Using the `auto-create-directory` Attribute

By default, if the destination directory does not exist, the respective destination directory and any non-existing parent directories are automatically created. To prevent that behavior, you can set the `auto-create-directory` attribute to `false`. This attribute applies to both the `directory` and the `directory-expression` attributes.

When using the `directory` attribute and `auto-create-directory` is `false`, the following change was made starting with Spring Integration 2.2:

Instead of checking for the existence of the destination directory when the adapter is initialized, this check is now performed for each message being processed.

Furthermore, if `auto-create-directory` is `true` and the directory was deleted between the processing of messages, the directory is re-created for each message being processed.

18.2.3. Dealing with Existing Destination Files

When you write files and the destination file already exists, the default behavior is to overwrite that target file. You can change this behavior by setting the `mode` attribute on the relevant file outbound components. The following options exist:

- `REPLACE` (Default)
- `REPLACE_IF_MODIFIED`
- `APPEND`
- `APPEND_NO_FLUSH`
- `FAIL`
- `IGNORE`

Spring Integration 2.2 introduced the `mode` attribute and the `APPEND`, `FAIL`, and `IGNORE` options.

`REPLACE`

If the target file already exists, it is overwritten. If the `mode` attribute is not specified, this is the default behavior when writing files.

`REPLACE_IF_MODIFIED`

If the target file already exists, it is overwritten only if the last modified timestamp differs from that of the source file. For `File` payloads, the payload `lastModified` time is compared to the existing file. For other payloads, the `FileHeaders.SET_MODIFIED` (`file_setModified`) header is compared to the existing file. If the header is missing or has a value that is not a `Number`, the file is always replaced.

`APPEND`

This mode lets you append message content to the existing file instead of creating a new file each time. Note that this attribute is mutually exclusive with the `temporary-file-suffix` attribute because, when it appends content to the existing file, the adapter no longer uses a temporary file. The file is closed after each message.

`APPEND_NO_FLUSH`

This option has the same semantics as `APPEND`, but the data is not flushed and the file is not closed after each message. This can provide a significant performance at the risk of data loss in the event of a failure. See Flushing Files When Using `APPEND_NO_FLUSH` for more information.

`FAIL`

If the target file exists, a `MessageHandlingException` is thrown.

`IGNORE`

If the target file exists, the message payload is silently ignored.

When using a temporary file suffix (the default is `.writing`), the `IGNORE` option applies if either the final file name or the temporary file name exists.

18.2.4. Flushing Files When Using `APPEND_NO_FLUSH`

The `APPEND_NO_FLUSH` mode was added in version 4.3. Using it can improve performance because the file is not closed after each message. However, this can cause data loss in the event of a failure.

Spring Integration provides several flushing strategies to mitigate this data loss:

- Use `flushInterval`. If a file is not written to for this period of time, it is automatically flushed. This is approximate and may be up to `1.33x` this time (with an average of `1.167x`).
- Send a message containing a regular expression to the message handler's `trigger` method. Files with absolute path names matching the pattern are flushed.
- Provide the handler with a custom `MessageFlushPredicate` implementation to modify the action taken when a message is sent to the `trigger` method.
- Invoke one of the handler's `flushIfNeeded` methods by passing in a custom `FileWritingMessageHandler.FlushPredicate` or `FileWritingMessageHandler.MessageFlushPredicate` implementation.

The predicates are called for each open file. See the Javadoc for these interfaces for more information. Note that, since version 5.0, the predicate methods provide another parameter: the time that the current file was first written to if new or previously closed.

When using `flushInterval`, the interval starts at the last write. The file is flushed only if it is idle for the interval. Starting with version 4.3.7, an additional property (`flushWhenIdle`) can be set to `false`, meaning that the interval starts with the first write to a previously flushed (or new) file.

18.2.5. File Timestamps

By default, the destination file's `lastModified` timestamp is the time when the file was created (except that an in-place rename retains the current timestamp). Starting with version 4.3, you can now configure `preserve-timestamp` (or `setPreserveTimestamp(true)` when using Java configuration). For `File` payloads, this transfers the timestamp from the inbound file to the outbound (regardless of whether a copy was required). For other payloads, if the `FileHeaders.SET_MODIFIED` header (`file_setModified`) is present, it is used to set the destination file's `lastModified` timestamp, as long as the header is a `Number`.

18.2.6. File Permissions

Starting with version 5.0, when writing files to a file system that supports Posix permissions, you can specify those permissions on the outbound channel adapter or gateway. The property is an integer and is usually supplied in the familiar octal format — for example, `0640`, meaning that the owner has read/write permissions, the group has read-only permission, and others have no access.

18.2.7. File Outbound Channel Adapter

The following example configures a file outbound channel adapter:

```
<int-file:outbound-channel-adapter id="filesOut" directory=
"${input.directory.property}"/>
```

The namespace-based configuration also supports a `delete-source-files` attribute. If set to `true`, it triggers the deletion of the original source files after writing to a destination. The default value for that flag is `false`. The following example shows how to set it to `true`:

```
<int-file:outbound-channel-adapter id="filesOut"
    directory="${output.directory}"
    delete-source-files="true"/>
```

The `delete-source-files` attribute has an effect only if the inbound message has a `File` payload or if the `FileHeaders.ORIGINAL_FILE` header value contains either the source `File` instance or a `String` representing the original file path.

Starting with version 4.2, the `FileWritingMessageHandler` supports an `append-new-line` option. If set to `true`, a new line is appended to the file after a message is written. The default attribute value is `false`. The following example shows how to use the `append-new-line` option:

```
<int-file:outbound-channel-adapter id="newlineAdapter"
    append-new-line="true"
    directory="${output.directory}"/>
```

18.2.8. Outbound Gateway

In cases where you want to continue processing messages based on the written file, you can use the `outbound-gateway` instead. It plays a role similar to that of the `outbound-channel-adapter`. However, after writing the file, it also sends it to the reply channel as the payload of a message.

The following example configures an outbound gateway:

```
<int-file:outbound-gateway id="mover" request-channel="moveInput"
    reply-channel="output"
    directory="${output.directory}"
    mode="REPLACE" delete-source-files="true"/>
```

As mentioned earlier, you can also specify the `mode` attribute, which defines the behavior of how to deal with situations where the destination file already exists. See Dealing with Existing Destination

Files for further details. Generally, when using the file outbound gateway, the result file is returned as the message payload on the reply channel.

This also applies when specifying the `IGNORE` mode. In that case the pre-existing destination file is returned. If the payload of the request message was a file, you still have access to that original file through the message header. See FileHeaders.ORIGINAL_FILE.

The 'outbound-gateway' works well in cases where you want to first move a file and then send it through a processing pipeline. In such cases, you may connect the file namespace's `inbound-channel-adapter` element to the `outbound-gateway` and then connect that gateway's `reply-channel` to the beginning of the pipeline.

If you have more elaborate requirements or need to support additional payload types as input to be converted to file content, you can extend the `FileWritingMessageHandler`, but a much better option is to rely on a `Transformer`.

18.2.9. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the inbound adapter with Java configuration:

```
@SpringBootApplication
@IntegrationComponentScan
public class FileWritingJavaApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
                        new SpringApplicationBuilder(FileWritingJavaApplication
.class)
                                .web(false)
                                .run(args);
            MyGateway gateway = context.getBean(MyGateway.class);
            gateway.writeToFile("foo.txt", new File(tmpDir.getRoot(),
"fileWritingFlow"), "foo");
    }

    @Bean
    @ServiceActivator(inputChannel = "writeToFileChannel")
    public MessageHandler fileWritingMessageHandler() {
         Expression directoryExpression = new SpelExpressionParser()
.parseExpression("headers.directory");
         FileWritingMessageHandler handler = new FileWritingMessageHandler
(directoryExpression);
         handler.setFileExistsMode(FileExistsMode.APPEND);
         return handler;
    }

    @MessagingGateway(defaultRequestChannel = "writeToFileChannel")
    public interface MyGateway {

        void writeToFile(@Header(FileHeaders.FILENAME) String fileName,
                        @Header(FileHeaders.FILENAME) File directory, String data);

    }
}
```

18.2.10. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the inbound adapter with the Java DSL:

```
@SpringBootApplication
public class FileWritingJavaApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
                new SpringApplicationBuilder(FileWritingJavaApplication.class)
                        .web(false)
                        .run(args);
        MessageChannel fileWritingInput = context.getBean("fileWritingInput",
MessageChannel.class);
        fileWritingInput.send(new GenericMessage<>("foo"));
    }

    @Bean
    public IntegrationFlow fileWritingFlow() {
        return IntegrationFlows.from("fileWritingInput")
                .enrichHeaders(h -> h.header(FileHeaders.FILENAME, "foo.txt")
                        .header("directory", new File(tmpDir.getRoot(),
"fileWritingFlow")))
                .handle(Files.outboundGateway(m -> m.getHeaders().get("directory"
)))
                .channel(MessageChannels.queue("fileWritingResultChannel"))
                .get();
    }

}
```

18.3. File Transformers

To transform data read from the file system to objects and the other way around, you need to do some work. Unlike `FileReadingMessageSource` and to a lesser extent `FileWritingMessageHandler`, you probably need your own mechanism to get the job done. For this, you can implement the `Transformer` interface. Alternatively, you can extend the `AbstractFilePayloadTransformer` for inbound messages. Spring Integration provides some obvious implementations.

See the Javadoc for the `Transformer` interface to see which Spring Integration classes implement it. Similarly, you can check the Javadoc for the `AbstractFilePayloadTransformer` class to see which Spring Integration classes extend it.

`FileToByteArrayTransformer` extends `AbstractFilePayloadTransformer` and transforms a `File` object into a `byte[]` by using Spring's `FileCopyUtils`. It is often better to use a sequence of transformers than to put all transformations in a single class. In that case the `File` to `byte[]` conversion might be a logical first step.

`FileToStringTransformer` extends `AbstractFilePayloadTransformer` convert a `File` object to a `String`. If nothing else, this can be useful for debugging (consider using it with a wire tap).

To configure file-specific transformers, you can use the appropriate elements from the file namespace, as the following example shows:

```
<int-file:file-to-bytes-transformer  input-channel="input" output-channel="output"
    delete-files="true"/>

<int-file:file-to-string-transformer input-channel="input" output-channel="output"
    delete-files="true" charset="UTF-8"/>
```

The `delete-files` option signals to the transformer that it should delete the inbound file after the transformation is complete. This is in no way a replacement for using an `AcceptOnceFileListFilter` when the `FileReadingMessageSource` is being used in a multi-threaded environment (such as when you use Spring Integration in general).

18.4. File Splitter

The `FileSplitter` was added in version 4.1.2, and its namespace support was added in version 4.2. The `FileSplitter` splits text files into individual lines, based on `BufferedReader.readLine()`. By default, the splitter uses an `Iterator` to emit lines one at a time as they are read from the file. Setting the `iterator` property to `false` causes it to read all the lines into memory before emitting them as messages. One use case for this might be if you want to detect I/O errors on the file before sending any messages containing lines. However, it is only practical for relatively short files.

Inbound payloads can be `File`, `String` (a `File` path), `InputStream`, or `Reader`. Other payload types are emitted unchanged.

The following listing shows all the possible attributes for `<int-file:splitter>`:

```
<int-file:splitter id="splitter" ①
    iterator=""                  ②
    markers=""                   ③
    markers-json=""              ④
    apply-sequence=""            ⑤
    requires-reply=""            ⑥
    charset=""                   ⑦
    first-line-as-header=""      ⑧
    input-channel=""             ⑨
    output-channel=""            ⑩
    send-timeout=""              ⑪
    auto-startup=""              ⑫
    order=""                     ⑬
    phase="" />                  ⑭
```

① The bean name of the splitter.

② Set to `true` (the default) to use an iterator or `false` to load the file into memory before sending lines.

③ Set to `true` to emit start-of-file and end-of-file marker messages before and after the file data. Markers are messages with `FileSplitter.FileMarker` payloads (with `START` and `END` values in the `mark` property). You might use markers when sequentially processing files in a downstream flow where some lines are filtered. They enable the downstream processing to know when a file has been completely processed. In addition, a `file_marker` header that contains `START` or `END` is added to these messages. The `END` marker includes a line count. If the file is empty, only `START` and `END` markers are emitted with `0` as the `lineCount`. The default is `false`. When `true`, `apply-sequence` is `false` by default. See also `markers-json` (the next attribute).

④ When `markers` is true, set this to `true` to have the `FileMarker` objects be converted to a JSON string. (Uses a `SimpleJsonSerializer` underneath).

⑤ Set to `false` to disable the inclusion of `sequenceSize` and `sequenceNumber` headers in messages. The default is `true`, unless `markers` is `true`. When `true` and `markers` is `true`, the markers are included in the sequencing. When `true` and `iterator` is `true`, the `sequenceSize` header is set to `0`, because the size is unknown.

⑥ Set to `true` to cause a `RequiresReplyException` to be thrown if there are no lines in the file. The default is `false`.

⑦ Set the charset name to be used when reading text data into `String` payloads. The default is the platform charset.

⑧ The header name for the first line to be carried as a header in the messages emitted for the remaining lines. Since version 5.0.

⑨ Set the input channel used to send messages to the splitter.

⑩ Set the output channel to which messages are sent.

⑪ Set the send timeout. Only applies if the `output-channel` can block — such as a full `QueueChannel`.

⑫ Set to `false` to disable automatically starting the splitter when the context is refreshed. The default is `true`.

⑬ Set the order of this endpoint if the `input-channel` is a `<publish-subscribe-channel/>`.

⑭ Set the startup phase for the splitter (used when `auto-startup` is `true`).

The `FileSplitter` also splits any text-based `InputStream` into lines. Starting with version 4.3, when used in conjunction with an FTP or SFTP streaming inbound channel adapter or an FTP or SFTP outbound gateway that uses the `stream` option to retrieve a file, the splitter automatically closes the session that supports the stream when the file is completely consumed See FTP Streaming Inbound Channel Adapter and SFTP Streaming Inbound Channel Adapter as well as FTP Outbound Gateway and SFTP Outbound Gateway for more information about these facilities.

When using Java configuration, an additional constructor is available, as the following example shows:

```
public FileSplitter(boolean iterator, boolean markers, boolean markersJson)
```

When `markersJson` is true, the markers are represented as a JSON string (using a `SimpleJsonSerializer`).

Version 5.0 introduced the `firstLineAsHeader` option to specify that the first line of content is a header (such as column names in a CSV file). The argument passed to this property is the header name under which the first line is carried as a header in the messages emitted for the remaining lines. This line is not included in the sequence header (if `applySequence` is true) nor in the `lineCount` associated with `FileMarker.END` . If a file contains only the header line, the file is treated as empty and, therefore, only `FileMarker` instances are emitted during splitting (if markers are enabled — otherwise, no messages are emitted). By default (if no header name is set), the first line is considered to be data and becomes the payload of the first emitted message.

If you need more complex logic about header extraction from the file content (not first line, not the whole content of the line, not one particular header, and so on), consider using header enricher ahead of the `FileSplitter`. Note that the lines that have been moved to the headers might be filtered downstream from the normal content process.

18.4.1. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure a file splitter with Java configuration:

```
@Splitter(inputChannel="toSplitter")
@Bean
public MessageHandler fileSplitter() {
    FileSplitter splitter = new FileSplitter(true, true);
    splitter.setApplySequence(true);
    splitter.setOutputChannel(outputChannel);
    return splitter;
}
```

18.4.2. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure a file splitter with the Java DSL:

```
@SpringBootApplication
public class FileSplitterApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FileSplitterApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public IntegrationFlow fileSplitterFlow() {
        return IntegrationFlows
            .from(Files.inboundAdapter(tmpDir.getRoot())
                 .filter(new ChainFileListFilter<File>()
                        .addFilter(new AcceptOnceFileListFilter<>())
                        .addFilter(new ExpressionFileListFilter<>(
                             new FunctionExpression<File>(f -> "foo.tmp".equals(f
.getName()))))))
            .split(Files.splitter()
                    .markers()
                    .charset(StandardCharsets.US_ASCII)
                    .firstLineAsHeader("fileHeader")
                    .applySequence(true))
            .channel(c -> c.queue("fileSplittingResultChannel"))
            .get();
    }

}
```

18.4.3. Idempotent Downstream Processing a Split File

When `apply-sequence` is true, the splitter adds the line number in the `SEQUENCE_NUMBER` header (when `markers` is true, the markers are counted as lines). The line number can be used with an Idempotent Receiver to avoid reprocessing lines after a restart.

For example:

```
@Bean
public ConcurrentMetadataStore store() {
    return new ZookeeperMetadataStore();
}

@Bean
public MetadataStoreSelector selector() {
    return new MetadataStoreSelector(
            message -> message.getHeaders().get(FileHeaders.ORIGINAL_FILE, File
.class)
                    .getAbsolutePath(),
            message -> message.getHeaders().get(IntegrationMessageHeaderAccessor
.SEQUENCE_NUMBER)
                    .toString(),
            store())
                    .compareValues(
                            (oldVal, newVal) -> Integer.parseInt(oldVal) <
Integer.parseInt(newVal));
}

@Bean
public IdempotentReceiverInterceptor idempotentReceiverInterceptor() {
    return new IdempotentReceiverInterceptor(selector());
}

@Bean
public IntegrationFlow flow() {
    ...
    .split(new FileSplitter())
    ...
    .handle("lineHandler", e -> e.advice(idempotentReceiverInterceptor()))
    ...
}
```

18.5. Remote Persistent File List Filters

Inbound and streaming inbound remote file channel adapters (`FTP`, `SFTP`, and other technologies) are configured with corresponding implementations of `AbstractPersistentFileListFilter` by default, configured with an in-memory `MetadataStore`. To run in a cluster, these can be replaced

with filters using a shared `MetadataStore` (see Metadata Store for more information). These filters are used to prevent fetching the same file multiple times (unless it's modified time changes). Starting with version 5.2, a file is added to the filter immediately before the file is fetched (and reversed if the fetch fails).

In the event of a catastrophic failure (such as power loss), it is possible that the file currently being fetched will remain in the filter and won't be re-fetched when restarting the application. In this case you would need to manually remove this file from the `MetadataStore`.

In previous versions, the files were filtered before any were fetched, meaning that several files could be in this state after a catastrophic failure.

In order to facilitate this new behavior, two new methods have been added to `FileListFilter`.

```
boolean accept(F file);

boolean supportsSingleFileFiltering();
```

If a filter returns `true` in `supportsSingleFileFiltering`, it **must** implement `accept()`.

If a remote filter does not support single file filtering (such as the `AbstractMarkerFilePresentFileListFilter`), the adapters revert to the previous behavior.

If multiple filters are in used (using a `CompositeFileListFilter` or `ChainFileListFilter`), then **all** of the delegate filters must support single file filtering in order for the composite filter to support it.

Chapter 19. FTP/FTPS Adapters

Spring Integration provides support for file transfer operations with FTP and FTPS.

The File Transfer Protocol (FTP) is a simple network protocol that lets you transfer files between two computers on the Internet. FTPS stands for “FTP over SSL”.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-ftp</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-ftp:5.3.0.RC1"
```

There are two actors when it comes to FTP communication: client and server. To transfer files with FTP or FTPS, you use a client that initiates a connection to a remote computer that is running an FTP server. After the connection is established, the client can choose to send or receive copies of files.

Spring Integration supports sending and receiving files over FTP or FTPS by providing three client-side endpoints: inbound channel adapter, outbound channel adapter, and outbound gateway. It also provides convenient namespace-based configuration options for defining these client components.

To use the FTP namespace, add the following to the header of your XML file:

```
xmlns:int-ftp="http://www.springframework.org/schema/integration/ftp"
xsi:schemaLocation="http://www.springframework.org/schema/integration/ftp
    https://www.springframework.org/schema/integration/ftp/spring-integration-
ftp.xsd"
```

19.1. FTP Session Factory

Spring Integration provides factories you can use to create FTP (or FTPS) sessions.

19.1.1. Default Factories

Starting with version 3.0, sessions are no longer cached by default. See FTP Session Caching.

Before configuring FTP adapters, you must configure an FTP session factory. You can configure the FTP Session Factory with a regular bean definition where the implementation class is `o.s.i.ftp.session.DefaultFtpSessionFactory`. The following example shows a basic configuration:

```
<bean id="ftpClientFactory"
    class="org.springframework.integration.ftp.session.DefaultFtpSessionFactory">
    <property name="host" value="localhost"/>
    <property name="port" value="22"/>
    <property name="username" value="kermit"/>
    <property name="password" value="frog"/>
    <property name="clientMode" value="0"/>
    <property name="fileType" value="2"/>
    <property name="bufferSize" value="100000"/>
</bean>
```

For FTPS connections, you can use `o.s.i.ftp.session.DefaultFtpsSessionFactory` instead.

The following example shows a complete configuration:

```
<bean id="ftpClientFactory"
    class="org.springframework.integration.ftp.session.DefaultFtpsSessionFactory">
    <property name="host" value="localhost"/>
    <property name="port" value="22"/>
    <property name="username" value="oleg"/>
    <property name="password" value="password"/>
    <property name="clientMode" value="1"/>
    <property name="fileType" value="2"/>
    <property name="useClientMode" value="true"/>
    <property name="cipherSuites" value="a,b.c"/>
    <property name="keyManager" ref="keyManager"/>
    <property name="protocol" value="SSL"/>
    <property name="trustManager" ref="trustManager"/>
    <property name="prot" value="P"/>
    <property name="needClientAuth" value="true"/>
    <property name="authValue" value="oleg"/>
    <property name="sessionCreation" value="true"/>
    <property name="protocols" value="SSL, TLS"/>
    <property name="implicit" value="true"/>
</bean>
```

If you experience connectivity problems and would like to trace session creation as well as see which sessions are polled, you can enable session tracing by setting the logger to the `TRACE` level (for example, `log4j.category.org.springframework.integration.file=TRACE`).

Now you need only inject these session factories into your adapters. The protocol (FTP or FTPS) that an adapter uses depends on the type of session factory that has been injected into the adapter.

A more practical way to provide values for FTP or FTPS session factories is to use Spring's property placeholder support (See docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-placeholderconfigurer).

19.2. Advanced Configuration

`DefaultFtpSessionFactory` provides an abstraction over the underlying client API, which (since Spring Integration 2.0) is Apache Commons Net. This spares you from the low-level configuration details of the `org.apache.commons.net.ftp.FTPClient`. Several common properties are exposed on the session factory (since version 4.0, this now includes `connectTimeout`, `defaultTimeout`, and `dataTimeout`). However, you sometimes need access to lower level `FTPClient` configuration to achieve more advanced configuration (such as setting the port range for active mode). For that purpose, `AbstractFtpSessionFactory` (the base class for all FTP Session Factories) exposes hooks, in the form of the two post-processing methods shown in the following listing:

```
/**
 * Will handle additional initialization after client.connect() method was
invoked,
 * but before any action on the client has been taken
 */
protected void postProcessClientAfterConnect(T t) throws IOException {
    // NOOP
}
/**
 * Will handle additional initialization before client.connect() method was
invoked.
 */
protected void postProcessClientBeforeConnect(T client) throws IOException {
    // NOOP
}
```

As you can see, there is no default implementation for these two methods. However, by extending `DefaultFtpSessionFactory`, you can override these methods to provide more advanced configuration of the `FTPClient`, as the following example shows:

```
public class AdvancedFtpSessionFactory extends DefaultFtpSessionFactory {

    protected void postProcessClientBeforeConnect(FTPClient ftpClient) throws
IOException {
        ftpClient.setActivePortRange(4000, 5000);
    }
}
```

19.2.1. FTPS and Shared SSLSession

When using FTP over SSL or TLS, some servers require the same `SSLSession` to be used on the control and data connections. This is to prevent "stealing" data connections. See scarybeastsecurity.blogspot.cz/2009/02/vsftpd-210-released.html for more information.

Currently, the Apache FTPSClient does not support this feature. See NET-408.

The following solution, courtesy of Stack Overflow, uses reflection on the `sun.security.ssl.SSLSessionContextImpl`, so it may not work on other JVMs. The stack overflow answer was submitted in 2015, and the solution has been tested by the Spring Integration team recently on JDK 1.8.0_112.

The following example shows how to create an FTPS session:

```
@Bean
public DefaultFtpsSessionFactory sf() {
    DefaultFtpsSessionFactory sf = new DefaultFtpsSessionFactory() {

        @Override
        protected FTPSClient createClientInstance() {
            return new SharedSSLFTPSClient();
        }

    };
    sf.setHost("...");
    sf.setPort(21);
    sf.setUsername("...");
    sf.setPassword("...");
    sf.setNeedClientAuth(true);
    return sf;
}

private static final class SharedSSLFTPSClient extends FTPSClient {

    @Override
    protected void _prepareDataSocket_(final Socket socket) throws IOException {
        if (socket instanceof SSLSocket) {
            // Control socket is SSL
```

```
            final SSLSession session = ((SSLSocket) _socket_).getSession();
            final SSLSessionContext context = session.getSessionContext();
            context.setSessionCacheSize(0); // you might want to limit the cache
            try {
                final Field sessionHostPortCache = context.getClass()
                        .getDeclaredField("sessionHostPortCache");
                sessionHostPortCache.setAccessible(true);
                final Object cache = sessionHostPortCache.get(context);
                final Method method = cache.getClass().getDeclaredMethod("put",
Object.class,
                        Object.class);
                method.setAccessible(true);
                String key = String.format("%s:%s", socket.getInetAddress()
.getHostName(),
                        String.valueOf(socket.getPort())).toLowerCase(Locale.ROOT);
                method.invoke(cache, key, session);
                key = String.format("%s:%s", socket.getInetAddress().getHostAddress(),
                        String.valueOf(socket.getPort())).toLowerCase(Locale.ROOT);
                method.invoke(cache, key, session);
            }
            catch (NoSuchFieldException e) {
                // Not running in expected JRE
                logger.warn("No field sessionHostPortCache in SSLSessionContext", e);
            }
            catch (Exception e) {
                // Not running in expected JRE
                logger.warn(e.getMessage());
            }
        }

    }

}
```

19.3. Delegating Session Factory

Version 4.2 introduced the `DelegatingSessionFactory`, which allows the selection of the actual session factory at runtime. Prior to invoking the FTP endpoint, call `setThreadKey()` on the factory to associate a key with the current thread. That key is then used to lookup the actual session factory to be used. You can clear the key by calling `clearThreadKey()` after use.

We added convenience methods so that you can easily do use a delegating session factory from a message flow.

The following example shows how to declare a delegating session factory:

```
<bean id="dsf" class=
"org.springframework.integration.file.remote.session.DelegatingSessionFactory">
    <constructor-arg>
        <bean class="o.s.i.file.remote.session.DefaultSessionFactoryLocator">
            <!-- delegate factories here -->
        </bean>
    </constructor-arg>
</bean>

<int:service-activator input-channel="in" output-channel="c1"
        expression="@dsf.setThreadKey(#root, headers['factoryToUse'])" />

<int-ftp:outbound-gateway request-channel="c1" reply-channel="c2" ... />

<int:service-activator input-channel="c2" output-channel="out"
        expression="@dsf.clearThreadKey(#root)" />
```

When you use session caching (see FTP Session Caching), each of the delegates should be cached. You cannot cache the `DelegatingSessionFactory` itself.

Starting with version 5.0.7, the `DelegatingSessionFactory` can be used in conjunction with a `RotatingServerAdvice` to poll multiple servers; see Inbound Channel Adapters: Polling Multiple Servers and Directories.

19.4. FTP Inbound Channel Adapter

The FTP inbound channel adapter is a special listener that connects to the FTP server and listens for the remote directory events (for example, new file created) at which point it initiates a file transfer. The following example shows how to configure an `inbound-channel-adapter`:

```
<int-ftp:inbound-channel-adapter id="ftpInbound"
    channel="ftpChannel"
    session-factory="ftpSessionFactory"
    auto-create-local-directory="true"
    delete-remote-files="true"
    filename-pattern="*.txt"
    remote-directory="some/remote/path"
    remote-file-separator="/"
    preserve-timestamp="true"
    local-filename-generator-expression="#this.toUpperCase() + '.a'"
    scanner="myDirScanner"
    local-filter="myFilter"
    temporary-file-suffix=".writing"
    max-fetch-size="-1"
    local-directory=".">
    <int:poller fixed-rate="1000"/>
</int-ftp:inbound-channel-adapter>
```

As the preceding configuration shows, you can configure an FTP inbound channel adapter by using the `inbound-channel-adapter` element while also providing values for various attributes, such as `local-directory`, `filename-pattern` (which is based on simple pattern matching, not regular expressions), and the reference to a `session-factory`.

By default, the transferred file carries the same name as the original file. If you want to override this behavior, you can set the `local-filename-generator-expression` attribute, which lets you provide a SpEL expression to generate the name of the local file. Unlike outbound gateways and adapters, where the root object of the SpEL evaluation context is a `Message`, this inbound adapter does not yet have the message at the time of evaluation, since that's what it ultimately generates with the transferred file as its payload. Consequently, the root object of the SpEL evaluation context is the original name of the remote file (a `String`).

The inbound channel adapter first retrieves the `File` object for a local directory and then emits each file according to the poller configuration. Starting with version 5.0, you can now limit the number of files fetched from the FTP server when new file retrievals are needed. This can be beneficial when the target files are very large or when you run in a clustered system with a persistent file list filter, discussed later. Use `max-fetch-size` for this purpose. A negative value (the default) means no limit and all matching files are retrieved. See Inbound Channel Adapters: Controlling Remote File Fetching for more information. Since version 5.0, you can also provide a custom `DirectoryScanner` implementation to the `inbound-channel-adapter` by setting the `scanner` attribute.

Starting with Spring Integration 3.0, you can specify the `preserve-timestamp` attribute (its default is `false`). When `true`, the local file's modified timestamp is set to the value retrieved from the server. Otherwise, it is set to the current time.

Starting with version 4.2, you can specify `remote-directory-expression` instead of `remote-directory`, letting you dynamically determine the directory on each poll — for example, `remote-directory-`

`expression="@myBean.determineRemoteDir()"`.

Starting with version 4.3, you can omit the `remote-directory` and `remote-directory-expression` attributes. They default to `null`. In this case, according to the FTP protocol, the client working directory is used as the default remote directory.

Sometimes, file filtering based on the simple pattern specified with the `filename-pattern` attribute might not suffice. If this is the case, you can use the `filename-regex` attribute to specify a regular expression (such as `filename-regex=".*\.test$"`). Also, if you need complete control, you can use the `filter` attribute and provide a reference to any custom implementation of the `o.s.i.file.filters.FileListFilter`, a strategy interface for filtering a list of files. This filter determines which remote files are retrieved. You can also combine a pattern-based filter with other filters (such as an `AcceptOnceFileListFilter` to avoid synchronizing files that have previously been fetched) by using a `CompositeFileListFilter`.

The `AcceptOnceFileListFilter` stores its state in memory. If you wish the state to survive a system restart, consider using the `FtpPersistentAcceptOnceFileListFilter` instead. This filter stores the accepted file names in an instance of the `MetadataStore` strategy (see Metadata Store). This filter matches on the filename and the remote modified time.

Since version 4.0, this filter requires a `ConcurrentMetadataStore`. When used with a shared data store (such as `Redis` with the `RedisMetadataStore`), it lets filter keys be shared across multiple application or server instances.

Starting with version 5.0, the `FtpPersistentAcceptOnceFileListFilter` with in-memory `SimpleMetadataStore` is applied by default for the `FtpInboundFileSynchronizer`. This filter is also applied with the `regex` or `pattern` option in the XML configuration as well as with `FtpInboundChannelAdapterSpec` in the Java DSL. Any other use cases can be managed with `CompositeFileListFilter` (or `ChainFileListFilter`).

The preceding discussion refers to filtering the files before retrieving them. Once the files have been retrieved, an additional filter is applied to the files on the file system. By default, this is an `AcceptOnceFileListFilter` which, as discussed earlier, retains state in memory and does not consider the file's modified time. Unless your application removes files after processing, the adapter will re-process the files on disk by default after an application restart.

Also, if you configure the `filter` to use a `FtpPersistentAcceptOnceFileListFilter` and the remote file timestamp changes (causing it to be re-fetched), the default local filter does not let this new file be processed.

For more information about this filter, and how it is used, see Remote Persistent File List Filters.

You can use the `local-filter` attribute to configure the behavior of the local file system filter. Starting with version 4.3.8, a `FileSystemPersistentAcceptOnceFileListFilter` is configured by default. This filter stores the accepted file names and modified timestamp in an instance of the `MetadataStore` strategy (see Metadata Store) and detects changes to the local file modified time. The default `MetadataStore` is a `SimpleMetadataStore`, which stores state in memory.

Since version 4.1.5, these filters have a new property (`flushOnUpdate`) that causes them to flush the metadata store on every update (if the store implements `Flushable`).

Further, if you use a distributed `MetadataStore` (such as Redis or GemFire), you can have multiple instances of the same adapter or application and be sure that each file is processed only once.

The actual local filter is a `CompositeFileListFilter` that contains the supplied filter and a pattern filter that prevents processing files that are in the process of being downloaded (based on the `temporary-file-suffix`). Files are downloaded with this suffix (the default is `.writing`), and the file is renamed to its final name when the transfer is complete, making it 'visible' to the filter.

The `remote-file-separator` attribute lets you configure a file separator character to use if the default '/' is not applicable for your particular environment.

See the schema for more details on these attributes.

You should also understand that the FTP inbound channel adapter is a polling consumer. Therefore, you must configure a poller (by using either a global default or a local sub-element). Once a file has been transferred, a message with a `java.io.File` as its payload is generated and sent to the channel identified by the `channel` attribute.

19.4.1. More on File Filtering and Incomplete Files

Sometimes the file that just appeared in the monitored (remote) directory is not complete. Typically, such a file is written with a temporary extension (such as `somefile.txt.writing`) and is then renamed once the writing process finishes. In most cases, you are only interested in files that are complete and would like to filter for only files that are complete. To handle these scenarios, you can use the filtering support provided by the `filename-pattern`, `filename-regex`, and `filter` attributes. The following example uses a custom filter implementation:

```
<int-ftp:inbound-channel-adapter
    channel="ftpChannel"
    session-factory="ftpSessionFactory"
    filter="customFilter"
    local-directory="file:/my_transfers">
    remote-directory="some/remote/path"
    <int:poller fixed-rate="1000"/>
</int-ftp:inbound-channel-adapter>

<bean id="customFilter" class="org.example.CustomFilter"/>
```

19.4.2. Poller Configuration Notes for the Inbound FTP Adapter

The job of the inbound FTP adapter consists of two tasks:

1. Communicate with a remote server in order to transfer files from a remote directory to a local directory.
2. For each transferred file, generate a message with that file as a payload and send it to the

channel identified by the 'channel' attribute. That is why they are called "'channel adapters'" rather than just "'adapters'". The main job of such an adapter is to generate a message to send to a message channel. Essentially, the second task takes precedence in such a way that, if your local directory already has one or more files, it first generates messages from those. Only when all local files have been processed does it initiate the remote communication to retrieve more files.

Also, when configuring a trigger on the poller, you should pay close attention to the `max-messages-per-poll` attribute. Its default value is `1` for all `SourcePollingChannelAdapter` instances (including FTP). This means that, as soon as one file is processed, it waits for the next execution time as determined by your trigger configuration. If you happened to have one or more files sitting in the `local-directory`, it would process those files before it would initiate communication with the remote FTP server. Also, if the `max-messages-per-poll` is set to `1` (the default), it processes only one file at a time with intervals as defined by your trigger, essentially working as “one-poll === one-file”.

For typical file-transfer use cases, you most likely want the opposite behavior: to process all the files you can for each poll and only then wait for the next poll. If that is the case, set `max-messages-per-poll` to -1. Then, on each poll, the adapter tries to generate as many messages as it possibly can. In other words, it processes everything in the local directory, and then it connects to the remote directory to transfer everything that is available there to be processed locally. Only then is the poll operation considered complete, and the poller waits for the next execution time.

You can alternatively set the 'max-messages-per-poll' value to a positive value that indicates the upward limit of messages to be created from files with each poll. For example, a value of `10` means that, on each poll, it tries to process no more than ten files.

19.4.3. Recovering from Failures

It is important to understand the architecture of the adapter. There is a file synchronizer that fetches the files and a `FileReadingMessageSource` that emits a message for each synchronized file. As discussed earlier, two filters are involved. The `filter` attribute (and patterns) refers to the remote (FTP) file list, to avoid fetching files that have already been fetched. The `local-filter` is used by the `FileReadingMessageSource` to determine which files are to be sent as messages.

The synchronizer lists the remote files and consults its filter. The files are then transferred. If an IO error occurs during file transfer, any files that have already been added to the filter are removed so that they are eligible to be re-fetched on the next poll. This only applies if the filter implements `ReversibleFileListFilter` (such as the `AcceptOnceFileListFilter`).

If, after synchronizing the files, an error occurs on the downstream flow processing a file, no automatic rollback of the filter occurs, so the failed file is not reprocessed by default.

If you wish to reprocess such files after a failure, you can use configuration similar to the following to facilitate the removal of the failed file from the filter:

```
<int-ftp:inbound-channel-adapter id="ftpAdapter"
        session-factory="ftpSessionFactory"
        channel="requestChannel"
        remote-directory-expression="'/ftpSource'"
        local-directory="file:myLocalDir"
        auto-create-local-directory="true"
        filename-pattern="*.txt">
    <int:poller fixed-rate="1000">
        <int:transactional synchronization-factory="syncFactory" />
    </int:poller>
</int-ftp:inbound-channel-adapter>

<bean id="acceptOnceFilter"
    class="org.springframework.integration.file.filters.AcceptOnceFileListFilter"
/>

<int:transaction-synchronization-factory id="syncFactory">
    <int:after-rollback expression="payload.delete()" />
</int:transaction-synchronization-factory>

<bean id="transactionManager"
    class="org.springframework.integration.transaction.PseudoTransactionManager"
/>
```

The preceding configuration works for any `ResettableFileListFilter`.

Starting with version 5.0, the inbound channel adapter can build sub-directories locally that correspond to the generated local file name. That can be a remote sub-path as well. To be able to read a local directory recursively for modification according to the hierarchy support, you can now supply an internal `FileReadingMessageSource` with a new `RecursiveDirectoryScanner` based on the `Files.walk()` algorithm. See `AbstractInboundFileSynchronizingMessageSource.setScanner()` for more information. Also, you can now switch the `AbstractInboundFileSynchronizingMessageSource` to the `WatchService`-based `DirectoryScanner` by using `setUseWatchService()` option. It is also configured for all the `WatchEventType` instances to react to any modifications in local directory. The reprocessing sample shown earlier is based on the built-in functionality of the `FileReadingMessageSource.WatchServiceDirectoryScanner` to perform `ResettableFileListFilter.remove()` when the file is deleted (`StandardWatchEventKinds.ENTRY_DELETE`) from the local directory. See `WatchServiceDirectoryScanner` for more information.

19.4.4. Configuring with Java Configuration

The following Spring Boot application show an example of how to configure the inbound adapter with Java configuration:

```
@SpringBootApplication
public class FtpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FtpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public SessionFactory<FTPFile> ftpSessionFactory() {
        DefaultFtpSessionFactory sf = new DefaultFtpSessionFactory();
        sf.setHost("localhost");
        sf.setPort(port);
        sf.setUsername("foo");
        sf.setPassword("foo");
        sf.setTestSession(true);
        return new CachingSessionFactory<FTPFile>(sf);
    }

    @Bean
    public FtpInboundFileSynchronizer ftpInboundFileSynchronizer() {
        FtpInboundFileSynchronizer fileSynchronizer = new
FtpInboundFileSynchronizer(ftpSessionFactory());
        fileSynchronizer.setDeleteRemoteFiles(false);
        fileSynchronizer.setRemoteDirectory("foo");
        fileSynchronizer.setFilter(new FtpSimplePatternFileListFilter("*.xml"));
        return fileSynchronizer;
    }

    @Bean
    @InboundChannelAdapter(channel = "ftpChannel", poller = @Poller(fixedDelay =
"5000"))
    public MessageSource<File> ftpMessageSource() {
        FtpInboundFileSynchronizingMessageSource source =
                new FtpInboundFileSynchronizingMessageSource
(ftpInboundFileSynchronizer());
        source.setLocalDirectory(new File("ftp-inbound"));
        source.setAutoCreateLocalDirectory(true);
        source.setLocalFilter(new AcceptOnceFileListFilter<File>());
        source.setMaxFetchSize(1);
        return source;
    }

    @Bean
    @ServiceActivator(inputChannel = "ftpChannel")
    public MessageHandler handler() {
        return new MessageHandler() {
```

```
            @Override
            public void handleMessage(Message<?> message) throws
MessagingException {
                System.out.println(message.getPayload());
            }

        };
    }

}
```

19.4.5. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the inbound adapter with the Java DSL:

```
@SpringBootApplication
public class FtpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FtpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public IntegrationFlow ftpInboundFlow() {
        return IntegrationFlows
            .from(Fpt.inboundAdapter(this.ftpSessionFactory)
                    .preserveTimestamp(true)
                    .remoteDirectory("foo")
                    .regexFilter(".*\\.txt$")
                    .localFilename(f -> f.toUpperCase() + ".a")
                    .localDirectory(new File("d:\\ftp_files")),
                e -> e.id("ftpInboundAdapter")
                    .autoStartup(true)
                    .poller(Pollers.fixedDelay(5000)))
            .handle(m -> System.out.println(m.getPayload()))
            .get();
    }
}
```

19.4.6. Dealing With Incomplete Data

See Dealing With Incomplete Data.

The `FtpSystemMarkerFilePresentFileListFilter` is provided to filter remote files that do not have a corresponding marker file on the remote system. See the Javadoc (and browse to the parent classes) for configuration information.

19.5. FTP Streaming Inbound Channel Adapter

Version 4.3 introduced the streaming inbound channel adapter. This adapter produces message with payloads of type `InputStream`, letting files be fetched without writing to the local file system. Since the session remains open, the consuming application is responsible for closing the session when the file has been consumed. The session is provided in the `closeableResource` header (`IntegrationMessageHeaderAccessor.CLOSEABLE_RESOURCE`). Standard framework components, such as the `FileSplitter` and `StreamTransformer`, automatically close the session. See File Splitter and Stream Transformer for more information about these components. The following example shows how to configure an `inbound-streaming-channel-adapter`:

```
<int-ftp:inbound-streaming-channel-adapter id="ftpInbound"
            channel="ftpChannel"
            session-factory="sessionFactory"
            filename-pattern="*.txt"
            filename-regex=".*\.txt"
            filter="filter"
            filter-expression="@myFilterBean.check(#root)"
            remote-file-separator="/"
            comparator="comparator"
            max-fetch-size="1"
            remote-directory-expression="'foo/bar'">
        <int:poller fixed-rate="1000" />
</int-ftp:inbound-streaming-channel-adapter>
```

Only one of `filename-pattern`, `filename-regex`, `filter`, or `filter-expression` is allowed.

Starting with version 5.0, by default, the `FtpStreamingMessageSource` adapter prevents duplicates for remote files with `FtpPersistentAcceptOnceFileListFilter` based on the in-memory `SimpleMetadataStore`. By default, this filter is also applied with the filename pattern (or regex). If you need to allow duplicates, you can use `AcceptAllFileListFilter`. Any other use cases can be handled by `CompositeFileListFilter` (or `ChainFileListFilter`). The Java configuration (later in the document) shows one technique to remove the remote file after processing to avoid duplicates.

For more information about the `FtpPersistentAcceptOnceFileListFilter`, and how it is used, see Remote Persistent File List Filters.

Use the `max-fetch-size` attribute to limit the number of files fetched on each poll when a fetch is necessary. Set it to `1` and use a persistent filter when running in a clustered environment. See Inbound Channel Adapters: Controlling Remote File Fetching for more information.

The adapter puts the remote directory and file name in the `FileHeaders.REMOTE_DIRECTORY` and `FileHeaders.REMOTE_FILE` headers, respectively. Starting with version 5.0, the `FileHeaders.REMOTE_FILE_INFO` header provides additional remote file information (represented in JSON by default). If you set the `fileInfoJson` property on the `FtpStreamingMessageSource` to `false`, the header contains an `FtpFileInfo` object. The `FTPFile` object provided by the underlying Apache Net library can be accessed by using the `FtpFileInfo.getFileInfo()` method. The `fileInfoJson` property is not available when you use XML configuration, but you can set it by injecting the `FtpStreamingMessageSource` into one of your configuration classes. See also Remote File Information.

Starting with version 5.1, the generic type of the `comparator` is `FTPFile`. Previously, it was `AbstractFileInfo<FTPFile>`. This is because the sort is now performed earlier in the processing, before filtering and applying `maxFetch`.

19.5.1. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the inbound adapter with Java configuration:

```
@SpringBootApplication
public class FtpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FtpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    @InboundChannelAdapter(channel = "stream")
    public MessageSource<InputStream> ftpMessageSource() {
        FtpStreamingMessageSource messageSource = new FtpStreamingMessageSource
(template());
        messageSource.setRemoteDirectory("ftpSource/");
        messageSource.setFilter(new AcceptAllFileListFilter<>());
        messageSource.setMaxFetchSize(1);
        return messageSource;
    }

    @Bean
    @Transformer(inputChannel = "stream", outputChannel = "data")
    public org.springframework.integration.transformer.Transformer transformer() {
        return new StreamTransformer("UTF-8");
    }

    @Bean
    public FtpRemoteFileTemplate template() {
        return new FtpRemoteFileTemplate(ftpSessionFactory());
    }

    @ServiceActivator(inputChannel = "data", adviceChain = "after")
    @Bean
    public MessageHandler handle() {
        return System.out::println;
    }

    @Bean
    public ExpressionEvaluatingRequestHandlerAdvice after() {
        ExpressionEvaluatingRequestHandlerAdvice advice = new
ExpressionEvaluatingRequestHandlerAdvice();
        advice.setOnSuccessExpression(
                "@template.remove(headers['file_remoteDirectory'] +
headers['file_remoteFile'])");
        advice.setPropagateEvaluationFailures(true);
        return advice;
    }

}
```

Notice that, in this example, the message handler downstream of the transformer has an advice that removes the remote file after processing.

19.6. Inbound Channel Adapters: Polling Multiple Servers and Directories

Starting with version 5.0.7, the `RotatingServerAdvice` is available; when configured as a poller advice, the inbound adapters can poll multiple servers and directories. Configure the advice and add it to the poller's advice chain as normal. A `DelegatingSessionFactory` is used to select the server see Delegating Session Factory for more information. The advice configuration consists of a list of `RotationPolicy.KeyDirectory` objects.

Example

```
@Bean
public RotatingServerAdvice advice() {
    List<RotationPolicy.KeyDirectory> keyDirectories = new ArrayList<>();
    keyDirectories.add(new RotationPolicy.KeyDirectory("one", "foo"));
    keyDirectories.add(new RotationPolicy.KeyDirectory("one", "bar"));
    keyDirectories.add(new RotationPolicy.KeyDirectory("two", "baz"));
    keyDirectories.add(new RotationPolicy.KeyDirectory("two", "qux"));
    keyDirectories.add(new RotationPolicy.KeyDirectory("three", "fiz"));
    keyDirectories.add(new RotationPolicy.KeyDirectory("three", "buz"));
    return new RotatingServerAdvice(delegatingSf(), keyDirectories);
}
```

This advice will poll directory `foo` on server `one` until no new files exist then move to directory `bar` and then directory `baz` on server `two`, etc.

This default behavior can be modified with the `fair` constructor arg:

fair

```
@Bean
public RotatingServerAdvice advice() {
    ...
    return new RotatingServerAdvice(delegatingSf(), keyDirectories, true);
}
```

In this case, the advice will move to the next server/directory regardless of whether the previous poll returned a file.

Alternatively, you can provide your own `RotationPolicy` to reconfigure the message source as needed:

policy

```
public interface RotationPolicy {

    void beforeReceive(MessageSource<?> source);

    void afterReceive(boolean messageReceived, MessageSource<?> source);

}
```

and

custom

```
@Bean
public RotatingServerAdvice advice() {
    return new RotatingServerAdvice(myRotationPolicy());
}
```

The `local-filename-generator-expression` attribute (`localFilenameGeneratorExpression` on the synchronizer) can now contain the `#remoteDirectory` variable. This allows files retrieved from different directories to be downloaded to similar directories locally:

```
@Bean
public IntegrationFlow flow() {
    return IntegrationFlows.from(Ftp.inboundAdapter(sf())
                    .filter(new FtpPersistentAcceptOnceFileListFilter(new
SimpleMetadataStore(), "rotate"))
                    .localDirectory(new File(tmpDir))
                    .localFilenameExpression("#remoteDirectory +
T(java.io.File).separator + #root")
                    .remoteDirectory("."),
                e -> e.poller(Pollers.fixedDelay(1).advice(advice())))
            .channel(MessageChannels.queue("files"))
            .get();
}
```

Do not configure a `TaskExecutor` on the poller when using this advice; see Conditional Pollers for Message Sources for more information.

19.7. Inbound Channel Adapters: Controlling Remote File Fetching

There are two properties that you should consider when you configure inbound channel adapters. `max-messages-per-poll`, as with all pollers, can be used to limit the number of messages emitted on each poll (if more than the configured value are ready). `max-fetch-size` (since version 5.0) can limit

the number of files retrieved from the remote server at one time.

The following scenarios assume the starting state is an empty local directory:

- `max-messages-per-poll=2` and `max-fetch-size=1`: The adapter fetches one file, emits it, fetches the next file, emits it, and then sleeps until the next poll.
- `max-messages-per-poll=2` and `max-fetch-size=2`): The adapter fetches both files and then emits each one.
- `max-messages-per-poll=2` and `max-fetch-size=4`: The adapter fetches up to four files (if available) and emits the first two (if there are at least two). The next two files are emitted on the next poll.
- `max-messages-per-poll=2` and `max-fetch-size` not specified: The adapter fetches all remote files and emits the first two (if there are at least two). The subsequent files are emitted on subsequent polls (two at a time). When all files are consumed, the remote fetch is attempted again, to pick up any new files.

When you deploy multiple instances of an application, we recommend a small `max-fetch-size`, to avoid one instance "grabbing" all the files and starving other instances.

Another use for `max-fetch-size` is if you want to stop fetching remote files but continue to process files that have already been fetched. Setting the `maxFetchSize` property on the `MessageSource` (programmatically, with JMX, or with a control bus) effectively stops the adapter from fetching more files but lets the poller continue to emit messages for files that have previously been fetched. If the poller is active when the property is changed, the change takes effect on the next poll.

Starting with version 5.1, the synchronizer can be provided with a `Comparator<FTPFile>`. This is useful when restricting the number of files fetched with `maxFetchSize`.

19.8. FTP Outbound Channel Adapter

The FTP outbound channel adapter relies on a `MessageHandler` implementation that connects to the FTP server and initiates an FTP transfer for every file it receives in the payload of incoming messages. It also supports several representations of a file, so you are not limited only to `java.io.File`-typed payloads. The FTP outbound channel adapter supports the following payloads:

- `java.io.File`: The actual file object
- `byte[]`: A byte array that represents the file contents
- `java.lang.String`: Text that represents the file contents
- `java.io.InputStream`: a stream of data to transfer to remote file
- `org.springframework.core.io.Resource`: a resource for data to transfer to remote file

The following example shows how to configure an `outbound-channel-adapter`:

```
<int-ftp:outbound-channel-adapter id="ftpOutbound"
    channel="ftpChannel"
    session-factory="ftpSessionFactory"
    charset="UTF-8"
    remote-file-separator="/"
    auto-create-directory="true"
    remote-directory-expression="headers['remote_dir']"
    temporary-remote-directory-expression="headers['temp_remote_dir']"
    filename-generator="fileNameGenerator"
    use-temporary-filename="true"
    chmod="600"
    mode="REPLACE"/>
```

The preceding configuration shows how you can configure an FTP outbound channel adapter by using the `outbound-channel-adapter` element while also providing values for various attributes, such as `filename-generator` (an implementation of the `o.s.i.file.FileNameGenerator` strategy interface), a reference to a `session-factory`, and other attributes. You can also see some examples of `*expression` attributes that let you use SpEL to configure settings such as `remote-directory-expression`, `temporary-remote-directory-expression`, and `remote-filename-generator-expression` (a SpEL alternative to `filename-generator`, shown in the preceding example). As with any component that allows the usage of SpEL, access to the payload and the message Headers is available through the 'payload' and 'headers' variables. See the schema for more details on the available attributes.

By default, if no file name generator is specified, Spring Integration uses `o.s.i.file.DefaultFileNameGenerator`. `DefaultFileNameGenerator` determines the file name based on the value of the `file_name` header (if it exists) in the `MessageHeaders`, or, if the payload of the Message is already a `java.io.File`, it uses the original name of that file.

Defining certain values (such as `remote-directory`) might be platform- or FTP server-dependent. For example, as was reported on forum.spring.io/showthread.php?p=333478&posted=1#post333478, on some platforms, you must add a slash to the end of the directory definition (for example, `remote-directory="/thing1/thing2/"` instead of `remote-directory="/thing1/thing2"`).

Starting with version 4.1, you can specify the `mode` when transferring the file. By default, an existing file is overwritten. The modes are defined by the `FileExistsMode` enumeration, which includes the following values:

- `REPLACE` (default)
- `REPLACE_IF_MODIFIED`
- `APPEND`
- `APPEND_NO_FLUSH`
- `IGNORE`

- `FAIL`

`IGNORE` and `FAIL` do not transfer the file. `FAIL` causes an exception to be thrown, while `IGNORE` silently ignores the transfer (although a `DEBUG` log entry is produced).

Version 5.2 introduced the `chmod` attribute, which you can use to change the remote file permissions after upload. You can use the conventional Unix octal format (for example, `600` allows read-write for the file owner only). When configuring the adapter using java, you can use `setChmodOctal("600")` or `setChmod(0600)`. Only applies if your FTP server supports the `SITE CHMOD` subcommand.

19.8.1. Avoiding Partially Written Files

One of the common problems that arises when dealing with file transfers is the possibility of processing a partial file. That is, a file might appear in the file system before its transfer is actually complete.

To deal with this issue, Spring Integration FTP adapters use a common algorithm: Files are transferred under a temporary name and then renamed once they are fully transferred.

By default, every file that is in the process of being transferred appears in the file system with an additional suffix, which, by default, is `.writing`. You can change this suffix by setting the `temporary-file-suffix` attribute.

However, there may be situations where you do not want to use this technique (for example, if the server does not permit renaming files). For situations like this, you can disable this feature by setting `use-temporary-file-name` to `false` (the default is `true`). When this attribute is `false`, the file is written with its final name and the consuming application needs some other mechanism to detect that the file is completely uploaded before accessing it.

19.8.2. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the outbound adapter with Java configuration:

```
@SpringBootApplication
@IntegrationComponentScan
public class FtpJavaApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
                    new SpringApplicationBuilder(FtpJavaApplication.class)
                        .web(false)
                        .run(args);
        MyGateway gateway = context.getBean(MyGateway.class);
        gateway.sendToFtp(new File("/foo/bar.txt"));
    }

    @Bean
    public SessionFactory<FTPFile> ftpSessionFactory() {
        DefaultFtpSessionFactory sf = new DefaultFtpSessionFactory();
        sf.setHost("localhost");
        sf.setPort(port);
        sf.setUsername("foo");
        sf.setPassword("foo");
        sf.setTestSession(true);
        return new CachingSessionFactory<FTPFile>(sf);
    }

    @Bean
    @ServiceActivator(inputChannel = "ftpChannel")
    public MessageHandler handler() {
        FtpMessageHandler handler = new FtpMessageHandler(ftpSessionFactory());
        handler.setRemoteDirectoryExpressionString("headers['remote-target-dir']"
);
        handler.setFileNameGenerator(new FileNameGenerator() {

            @Override
            public String generateFileName(Message<?> message) {
                 return "handlerContent.test";
            }

        });
        return handler;
    }

    @MessagingGateway
    public interface MyGateway {

         @Gateway(requestChannel = "toFtpChannel")
         void sendToFtp(File file);

    }
}
```

19.8.3. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the outbound adapter using the Java DSL:

```
@SpringBootApplication
@IntegrationComponentScan
public class FtpJavaApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
            new SpringApplicationBuilder(FtpJavaApplication.class)
                .web(false)
                .run(args);
        MyGateway gateway = context.getBean(MyGateway.class);
        gateway.sendToFtp(new File("/foo/bar.txt"));
    }

    @Bean
    public SessionFactory<FTPFile> ftpSessionFactory() {
        DefaultFtpSessionFactory sf = new DefaultFtpSessionFactory();
        sf.setHost("localhost");
        sf.setPort(port);
        sf.setUsername("foo");
        sf.setPassword("foo");
        sf.setTestSession(true);
        return new CachingSessionFactory<FTPFile>(sf);
    }

    @Bean
    public IntegrationFlow ftpOutboundFlow() {
        return IntegrationFlows.from("toFtpChannel")
                .handle(Ftp.outboundAdapter(ftpSessionFactory(), FileExistsMode
.FAIL)
                        .useTemporaryFileName(false)
                        .fileNameExpression("headers['" + FileHeaders.FILENAME +
"']")
                        .remoteDirectory(this.ftpServer.getTargetFtpDirectory()
.getName())
                ).get();
    }

    @MessagingGateway
    public interface MyGateway {

         @Gateway(requestChannel = "toFtpChannel")
         void sendToFtp(File file);

    }

}
```

19.9. FTP Outbound Gateway

The FTP outbound gateway provides a limited set of commands to interact with a remote FTP or FTPS server. The supported commands are:

- `ls` (list files)
- `nlst` (list file names)
- `get` (retrieve file)
- `mget` (retrieve file(s))
- `rm` (remove file(s))
- `mv` (move/rename file)
- `put` (send file)
- `mput` (send multiple files)

19.9.1. Using the `ls` Command

`ls` lists remote files and supports the following options:

- `-1`: Retrieve a list of file names. The default is to retrieve a list of `FileInfo` objects.
- `-a`: Include all files (including those starting with '.')
- `-f`: Do not sort the list
- `-dirs`: Include directories (they are excluded by default)
- `-links`: Include symbolic links (they are excluded by default)
- `-R`: List the remote directory recursively

In addition, filename filtering is provided, in the same manner as the `inbound-channel-adapter`. See FTP Inbound Channel Adapter.

The message payload resulting from an `ls` operation is a list of file names or a list of `FileInfo` objects. These objects provide information such as modified time, permissions, and other details.

The remote directory that the `ls` command acted on is provided in the `file_remoteDirectory` header.

When using the recursive option (`-R`), the `fileName` includes any subdirectory elements, representing a relative path to the file (relative to the remote directory). If the `-dirs` option is included, each recursive directory is also returned as an element in the list. In this case, it is recommended that you not use the `-1` option, because you would not be able to distinguish files from directories, which you can do with the `FileInfo` objects.

Starting with version 4.3, the `FtpSession` supports `null` for the `list()` and `listNames()` methods. Therefore, you can omit the `expression` attribute. For convenience, Java configuration has two constructors that do not have an `expression` argument. or `LS`, `NLST`, `PUT` and `MPUT` commands, `null` is treated as the client working directory, according to the FTP protocol. All other commands must be supplied with the `expression` to evaluate the remote path against the request message. You can set the working directory with the `FTPClient.changeWorkingDirectory()` function when you extend the

`DefaultFtpSessionFactory` and implement the `postProcessClientAfterConnect()` callback.

19.9.2. Using the `nlst` Command

Version 5 introduced support for the `nlst` command.

`nlst` lists remote file names and supports only one option:

- `-f`: Do not sort the list

The message payload resulting from an `nlst` operation is a list of file names.

The remote directory that the `nlst` command acted on is provided in the `file_remoteDirectory` header.

Unlike the `-1` option for the `ls` command, which uses the `LIST` command, the `nlst` command sends an `NLST` command to the target FTP server. This command is useful when the server does not support `LIST` (due to security restrictions, for example). The result of the `nlst` operation is the names without other detail. Therefore, the framework cannot determine if an entity is a directory, to perform filtering or recursive listing, for example.

19.9.3. Using the `get` Command

`get` retrieves a remote file. It supports the following option:

- `-P`: Preserve the timestamp of the remote file.
- `-stream`: Retrieve the remote file as a stream.
- `-D`: Delete the remote file after successful transfer. The remote file is not deleted if the transfer is ignored, because the `FileExistsMode` is `IGNORE` and the local file already exists.

The `file_remoteDirectory` header provides the remote directory name, and the `file_remoteFile` header provides the file name.

The message payload resulting from a `get` operation is a `File` object that represents the retrieved file or an `InputStream` when you use the `-stream` option. The `-stream` option allows retrieving the file as a stream. For text files, a common use case is to combine this operation with a file splitter or a stream transformer. When consuming remote files as streams, you are responsible for closing the `Session` after the stream is consumed. For convenience, the `Session` is provided in the `closeableResource` header, which you can access with a convenience method on `IntegrationMessageHeaderAccessor` The following example shows how to use the convenience method:

```
Closeable closeable = new IntegrationMessageHeaderAccessor(message)
.getCloseableResource();
if (closeable != null) {
    closeable.close();
}
```

Framework components such as the file splitter and the stream transformer automatically close the session after the data is transferred.

The following example shows how to consume a file as a stream:

```
<int-ftp:outbound-gateway session-factory="ftpSessionFactory"
                          request-channel="inboundGetStream"
                          command="get"
                          command-options="-stream"
                          expression="payload"
                          remote-directory="ftpTarget"
                          reply-channel="stream" />

<int-file:splitter input-channel="stream" output-channel="lines" />
```

If you consume the input stream in a custom component, you must close the `Session`. You can do so either in your custom code or by routing a copy of the message to a `service-activator` and using SpEL, as the following example shows:

```
<int:service-activator input-channel="closeSession"
    expression="headers['closeableResource'].close()" />
```

19.9.4. Using the `mget` Command

`mget` retrieves multiple remote files based on a pattern and supports the following options:

- `-P`: Preserve the timestamps of the remote files.
- `-R`: Retrieve the entire directory tree recursively.
- `-x`: Throw an exception if no files match the pattern (otherwise an empty list is returned).
- `-D`: Delete each remote file after successful transfer. The remote file is not deleted if the transfer is ignored, because the `FileExistsMode` is `IGNORE` and the local file already exists.

The message payload resulting from an `mget` operation is a `List<File>` object (that is, a `List` of `File` objects, each representing a retrieved file).

Starting with version 5.0, if the `FileExistsMode` is `IGNORE`, the payload of the output message no longer contains files that were not fetched due to the file already existing. Previously, the list contained all files, including those that already existed.

The expression used to determine the remote path should produce a result that ends with `/` - e.g. `somedir/` will fetch the complete tree under `somedir`.

Starting with version 5.0, a recursive `mget`, combined with the new

`FileExistsMode.REPLACE_IF_MODIFIED` mode, can be used to periodically synchronize an entire remote directory tree locally. This mode replaces the local file's last modified timestamp with the remote timestamp, regardless of the `-P` (preserve timestamp) option.

Using recursion (`-R`)

The pattern is ignored, and `*` is assumed. By default, the entire remote tree is retrieved. However, files in the tree can be filtered, by providing a `FileListFilter`. Directories in the tree can also be filtered this way. A `FileListFilter` can be provided by reference, by `filename-pattern`, or by `filename-regex` attributes. For example, `filename-regex="(subDir|.*1.txt)"` retrieves all files ending with `1.txt` in the remote directory and the `subDir` child directory. However, the next example shows an alternative, which version 5.0 made available.

If a subdirectory is filtered, no additional traversal of that subdirectory is performed.

The `-dirs` option is not allowed (the recursive `mget` uses the recursive `ls` to obtain the directory tree, so the directories themselves cannot be included in the list).

Typically, you would use the `#remoteDirectory` variable in the `local-directory-expression` so that the remote directory structure is retained locally.

Starting with version 5.0, the `FtpSimplePatternFileListFilter` and `FtpRegexPatternFileListFilter` can be configured to always pass directories by setting the `alwaysAcceptDirectories` property to `true`. Doing so allows recursion for a simple pattern, as the following examples show:

```
<bean id="starDotTxtFilter"
        class=
"org.springframework.integration.ftp.filters.FtpSimplePatternFileListFilter">
    <constructor-arg value="*.txt" />
    <property name="alwaysAcceptDirectories" value="true" />
</bean>

<bean id="dotStarDotTxtFilter"
            class=
"org.springframework.integration.ftp.filters.FtpRegexPatternFileListFilter">
    <constructor-arg value="^.*\.txt$" />
    <property name="alwaysAcceptDirectories" value="true" />
</bean>
```

Once you have defined filters such as those in the preceding example, you can use one by setting the `filter` property on the gateway.

See also Outbound Gateway Partial Success (`mget` and `mput`).

19.9.5. Using the `put` Command

The `put` commad sends a file to the remote server. The payload of the message can be a `java.io.File`, a `byte[]`, or a `String`. A `remote-filename-generator` (or expression) is used to name the remote file. Other available attributes include `remote-directory`, `temporary-remote-directory`, and their `*-expression` equivalents: `use-temporary-file-name` and `auto-create-directory`. See the schema documentation for more information.

The message payload resulting from a `put` operation is a `String` that represents the full path of the file on the server after transfer.

Version 5.2 introduced the `chmod` attribute, which changes the remote file permissions after upload. You can use the conventional Unix octal format (for example, `600` allows read-write for the file owner only). When configuring the adapter using java, you can use `setChmod(0600)`. Only applies if your FTP server supports the `SITE CHMOD` subcommand.

Using the `mput` Command

The `mput` sends multiple files to the server and supports only one option:

- `-R`: Recursive. Send all files (possibly filtered) in the directory and its subdirectories.

The message payload must be a `java.io.File` (or `String`) that represents a local directory. Since version 5.1, a collection of `File` or `String` is also supported.

This command supports the same attributes as the `put` command. In addition, files in the local directory can be filtered with one of `mput-pattern`, `mput-regex`, `mput-filter`, or `mput-filter-expression`. The filter works with recursion, as long as the subdirectories themselves pass the filter. Subdirectories that do not pass the filter are not recursed.

The message payload resulting from an `mget` operation is a `List<String>` object (that is, a `List` of remote file paths that result from the transfer).

See also Outbound Gateway Partial Success (`mget` and `mput`).

Version 5.2 introduced the `chmod` attribute, which lets you change the remote file permissions after upload. You can use the conventional Unix octal format (for example, `600` allows read-write for the file owner only). When configuring the adapter with Java, you can use `setChmodOctal("600")` or `setChmod(0600)`. Only applies if your FTP server supports the `SITE CHMOD` subcommand.

19.9.6. Using the `rm` Command

The `rm` command removes files.

The `rm` command has no options.

The message payload resulting from an `rm` operation is `Boolean.TRUE` if the remove was successful or `Boolean.FALSE` otherwise. The `file_remoteDirectory` header provides the remote directory, and the `file_remoteFile` header provides the file name.

19.9.7. Using the `mv` Command

The `mv` command moves files

The `mv` command has no options.

The `expression` attribute defines the “from” path and the `rename-expression` attribute defines the “to” path. By default, the `rename-expression` is `headers['file_renameTo']`. This expression must not evaluate to null or an empty `String`. If necessary, any necessary remote directories are created. The payload of the result message is `Boolean.TRUE`. The `file_remoteDirectory` header provides the original remote directory, and `file_remoteFile` header provides the file name. The new path is in the `file_renameTo` header.

19.9.8. Additional Information about FTP Outbound Gateway Commands

The `get` and `mget` commands support the `local-filename-generator-expression` attribute. It defines a SpEL expression to generate the name of local files during the transfer. The root object of the evaluation context is the request message. The `remoteFileName` variable, which is particularly useful for `mget`, is also available — for example, `local-filename-generator-expression="#remoteFileName.toUpperCase() + headers.something"`.

The `get` and `mget` commands support the `local-directory-expression` attribute. It defines a SpEL expression to generate the name of local directories during the transfer. The root object of the evaluation context is the request message but. The `remoteDirectory` variable, which is particularly useful for `mget`, is also available — for example: `local-directory-expression="'/tmp/local/' + #remoteDirectory.toUpperCase() + headers.something"`. This attribute is mutually exclusive with the `local-directory` attribute.

For all commands, the 'expression' property of the gateway provides the path on which the command acts. For the `mget` command, the expression might evaluate to '**', meaning to retrieve all files, or 'somedirectory**/', and so on.

The following example shows a gateway configured for an `ls` command:

```
<int-ftp:outbound-gateway id="gateway1"
    session-factory="ftpSessionFactory"
    request-channel="inbound1"
    command="ls"
    command-options="-1"
    expression="payload"
    reply-channel="toSplitter"/>
```

The payload of the message sent to the `toSplitter` channel is a list of `String` objects that each contain the name of a file. If the `command-options` attribute was omitted, it holds `FileInfo` objects. It uses space-delimited options — for example, `command-options="-1 -dirs -links"`.

Starting with version 4.2, the `GET`, `MGET`, `PUT` and `MPUT` commands support a `FileExistsMode` property

(`mode` when using the namespace support). This affects the behavior when the local file exists (`GET` and `MGET`) or the remote file exists (`PUT` and `MPUT`). Supported modes are `REPLACE`, `APPEND`, `FAIL`, and `IGNORE`. For backwards compatibility, the default mode for `PUT` and `MPUT` operations is `REPLACE`. For `GET` and `MGET` operations, the default is `FAIL`.

Starting with version 5.0, the `setWorkingDirExpression()` (`working-dir-expression` in XML) option is provided on the `FtpOutboundGateway` (`<int-ftp:outbound-gateway>` in XML). It lets you change the client working directory at runtime. The expression is evaluated against the request message. The previous working directory is restored after each gateway operation.

19.9.9. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the outbound gateway with Java configuration:

```
@SpringBootApplication
public class FtpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FtpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public SessionFactory<FTPFile> ftpSessionFactory() {
        DefaultFtpSessionFactory sf = new DefaultFtpSessionFactory();
        sf.setHost("localhost");
        sf.setPort(port);
        sf.setUsername("foo");
        sf.setPassword("foo");
        sf.setTestSession(true);
        return new CachingSessionFactory<FTPFile>(sf);
    }

    @Bean
    @ServiceActivator(inputChannel = "ftpChannel")
    public MessageHandler handler() {
        FtpOutboundGateway ftpOutboundGateway =
                        new FtpOutboundGateway(ftpSessionFactory(), "ls",
"'my_remote_dir/'");
        ftpOutboundGateway.setOutputChannelName("lsReplyChannel");
        return ftpOutboundGateway;
    }

}
```

19.9.10. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the outbound gateway with the Java DSL:

```
@SpringBootApplication
public class FtpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FtpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public SessionFactory<FTPFile> ftpSessionFactory() {
        DefaultFtpSessionFactory sf = new DefaultFtpSessionFactory();
        sf.setHost("localhost");
        sf.setPort(port);
        sf.setUsername("foo");
        sf.setPassword("foo");
        sf.setTestSession(true);
        return new CachingSessionFactory<FTPFile>(sf);
    }

    @Bean
    public FtpOutboundGatewaySpec ftpOutboundGateway() {
        return Ftp.outboundGateway(ftpSessionFactory(),
            AbstractRemoteFileOutboundGateway.Command.MGET, "payload")
            .options(AbstractRemoteFileOutboundGateway.Option.RECURSIVE)
            .regexFileNameFilter("(subFtpSource|.*1.txt)")
            .localDirectoryExpression("'localDirectory/' + #remoteDirectory")
            .localFilenameExpression("#remoteFileName.replaceFirst('ftpSource',
'localTarget')");
    }

    @Bean
    public IntegrationFlow ftpMGetFlow(AbstractRemoteFileOutboundGateway<FTPFile>
ftpOutboundGateway) {
        return f -> f
            .handle(ftpOutboundGateway)
            .channel(c -> c.queue("remoteFileOutputChannel"));
    }

}
```

19.9.11. Outbound Gateway Partial Success (mget and mput)

When you perform operations on multiple files (by using mget and mput), an exception can occur some time after one or more files have been transferred. In this case (starting with version 4.2), a PartialSuccessException is thrown. As well as the usual MessagingException properties (failedMessage and cause), this exception has two additional properties:

- partialResults: The successful transfer results.
- derivedInput: The list of files generated from the request message (for example, local files to transfer for an mput).

These attributes let you determine which files were successfully transferred and which were not.

In the case of a recursive mput, the PartialSuccessException may have nested PartialSuccessException occurrences.

Consider the following directory structure:

```
root/
|- file1.txt
|- subdir/
   | - file2.txt
   | - file3.txt
|- zoo.txt
```

If the exception occurs on file3.txt, the PartialSuccessException thrown by the gateway has derivedInput of file1.txt, subdir, and zoo.txt and partialResults of file1.txt. Its cause is another PartialSuccessException with derivedInput of file2.txt and file3.txt and partialResults of file2.txt.

19.10. FTP Session Caching

Starting with Spring Integration 3.0, sessions are no longer cached by default. The cache-sessions attribute is no longer supported on endpoints. You must use a CachingSessionFactory (shown in the next example) if you wish to cache sessions.

In versions prior to 3.0, the sessions were automatically cached by default. A cache-sessions attribute was available for disabling the auto caching, but that solution did not provide a way to configure other session caching attributes. For example, you could not limit the number of sessions created. To support that requirement and other configuration options, a CachingSessionFactory was added. It provides sessionCacheSize and sessionWaitTimeout properties. The sessionCacheSize property controls how many active sessions the factory maintains in its cache (the default is unbounded). If the sessionCacheSize threshold has been reached, any attempt to acquire another session blocks until either one of the cached sessions becomes available or until the wait time for a session expires (the default wait time is Integer.MAX_VALUE). The sessionWaitTimeout property configures that value.

If you want your sessions to be cached, configure your default session factory as described earlier and then wrap it in an instance of `CachingSessionFactory`, where you can provide those additional properties. The following example shows how to do so:

```
<bean id="ftpSessionFactory" class="o.s.i.ftp.session.DefaultFtpSessionFactory">
    <property name="host" value="localhost"/>
</bean>

<bean id="cachingSessionFactory" class=
"o.s.i.file.remote.session.CachingSessionFactory">
    <constructor-arg ref="ftpSessionFactory"/>
    <constructor-arg value="10"/>
    <property name="sessionWaitTimeout" value="1000"/>
</bean>
```

The preceding example shows a `CachingSessionFactory` created with the `sessionCacheSize` set to `10` and the `sessionWaitTimeout` set to one second (its value is in milliseconds).

Starting with Spring Integration 3.0, the `CachingConnectionFactory` provides a `resetCache()` method. When invoked, all idle sessions are immediately closed and in-use sessions are closed when they are returned to the cache. New requests for sessions establish new sessions as necessary.

Starting with version 5.1, the `CachingSessionFactory` has a new property `testSession`. When true, the session will be tested by sending a NOOP command to ensure it is still active; if not, it will be removed from the cache; a new session is created if no active sessions are in the cache.

19.11. Using `RemoteFileTemplate`

Starting with Spring Integration 3.0, a new abstraction is provided over the `FtpSession` object. The template provides methods to send, retrieve (as an `InputStream`), remove, and rename files. In addition an `execute` method is provided allowing the caller to execute multiple operations on the session. In all cases, the template takes care of reliably closing the session. For more information, see the Javadoc for `RemoteFileTemplate`. There is a subclass for FTP: `FtpRemoteFileTemplate`.

Version 4.1 added added additional methods, including `getClientInstance()`, which provides access to the underlying `FTPClient` and thus gives you access to low-level APIs.

Not all FTP servers properly implement the `STAT <path>` command. Some return a positive result for a non-existent path. The `NLST` command reliably returns the name when the path is a file and it exists. However, this does not support checking that an empty directory exists since `NLST` always returns an empty list when the path is a directory. Since the template does not know whether the path represents a directory, it has to perform additional checks when the path does not appear to exist (when using `NLST`). This adds overhead, requiring several requests to the server. Starting with version 4.1.9, the `FtpRemoteFileTemplate` provides the `FtpRemoteFileTemplate.ExistsMode` property, which has the following options:

- `STAT`: Perform the `STAT` FTP command (`FTPClient.getStatus(path)`) to check the path existence.

This is the default and requires that your FTP server properly support the `STAT` command (with a path).

- `NLST`: Perform the `NLST` FTP command — `FTPClient.listName(path)`. Use this if you are testing for a path that is a full path to a file. It does not work for empty directories.
- `NLST_AND_DIRS`: Perform the `NLST` command first and, if it returns no files, fall back to a technique that temporarily switches the working directory by using `FTPClient.changeWorkingDirectory(path)`. See `FtpSession.exists()` for more information.

Since we know that the `FileExistsMode.FAIL` case is always only looking for a file (and not a directory), we safely use `NLST` mode for the `FtpMessageHandler` and `FtpOutboundGateway` components.

For any other cases, the `FtpRemoteFileTemplate` can be extended to implement custom logic in the overridden `exist()` method.

Starting with version 5.0, the new `RemoteFileOperations.invoke(OperationsCallback<F, T> action)` method is available. This method lets several `RemoteFileOperations` calls be called in the scope of the same, thread-bounded, `Session`. This is useful when you need to perform several high-level operations of the `RemoteFileTemplate` as one unit of work. For example, `AbstractRemoteFileOutboundGateway` uses it with the `mput` command implementation, where we perform a `put` operation for each file in the provided directory and recursively for its sub-directories. See the Javadoc for more information.

19.12. Using `MessageSessionCallback`

Starting with Spring Integration 4.2, you can use a `MessageSessionCallback<F, T>` implementation with the `<int-ftp:outbound-gateway/>` (`FtpOutboundGateway` in Java) to perform any operations on the `Session<FTPFile>` with the `requestMessage` context. It can be used for any non-standard or low-level FTP operations and allows access from an integration flow definition and functional interface (Lambda) implementation injection, as the following example shows:

```
@Bean
@ServiceActivator(inputChannel = "ftpChannel")
public MessageHandler ftpOutboundGateway(SessionFactory<FTPFile> sessionFactory) {
    return new FtpOutboundGateway(sessionFactory,
         (session, requestMessage) -> session.list(requestMessage.getPayload()));
}
```

Another example might be to pre- or post-process the file data being sent or retrieved.

When using XML configuration, the `<int-ftp:outbound-gateway/>` provides a `session-callback` attribute to let you specify the `MessageSessionCallback` bean name.

The `session-callback` is mutually exclusive with the `command` and `expression` attributes. When configuring with Java, different constructors are available in the `FtpOutboundGateway` class.

19.13. Apache Mina FTP Server Events

The `ApacheMinaFtplet`, added in version 5.2, listens for certain Apache Mina FTP server events and publishes them as `ApplicationEvent` s which can be received by any `ApplicationListener` bean, `@EventListener` bean method, or Event Inbound Channel Adapter.

Currently supported events are:

- `SessionOpenedEvent` - a client session was opened
- `DirectoryCreatedEvent` - a directory was created
- `FileWrittenEvent` - a file was written to
- `PathMovedEvent` - a file or directory was renamed
- `PathRemovedEvent` - a file or directory was removed
- `SessionClosedEvent` - the client has disconnected

Each of these is a subclass of `ApacheMinaFtpEvent`; you can configure a single listener to receive all of the event types. The `source` property of each event is a `FtpSession`, from which you can obtain information such as the client address; a convenient `getSession()` method is provided on the abstract event.

Events other than session open/close have another property `FtpRequest` which has properties such as the command and arguments.

To configure the server with the listener (which must be a Spring bean), add it to the server factory:

```
FtpServerFactory serverFactory = new FtpServerFactory();
...
ListenerFactory factory = new ListenerFactory();
...
serverFactory.addListener("default", factory.createListener());
serverFactory.setFtplets(new HashMap<>(Collections.singletonMap("springFtplet",
apacheMinaFtpletBean)));
server = serverFactory.createServer();
server.start();
```

To consume these events using a Spring Integration event adapter:

```
@Bean
public ApplicationEventListeningMessageProducer eventsAdapter() {
    ApplicationEventListeningMessageProducer producer =
        new ApplicationEventListeningMessageProducer();
    producer.setEventTypes(ApacheMinaFtpEvent.class);
    producer.setOutputChannel(eventChannel());
    return producer;
}
```

19.14. Remote File Information

Starting with version 5.2, the `FtpStreamingMessageSource` (FTP Streaming Inbound Channel Adapter), `FtpInboundFileSynchronizingMessageSource` (FTP Inbound Channel Adapter) and "read"-commands of the `FtpOutboundGateway` (FTP Outbound Gateway) provide additional headers in the message to produce with an information about the remote file:

- `FileHeaders.REMOTE_HOST_PORT` - the host:port pair the remote session has been connected to during file transfer operation;
- `FileHeaders.REMOTE_DIRECTORY` - the remote directory the operation has been performed;
- `FileHeaders.REMOTE_FILE` - the remote file name; applicable only for single file operations.

Since the `FtpInboundFileSynchronizingMessageSource` doesn't produce messages against remote files, but using a local copy, the `AbstractInboundFileSynchronizer` stores an information about remote file in the `MetadataStore` (which can be configured externally) in the URI style (`protocol://host:port/remoteDirectory#remoteFileName`) during synchronization operation. This metadata is retrieved by the `FtpInboundFileSynchronizingMessageSource` when local file is polled. When local file is deleted, it is recommended to remove its metadata entry. The `AbstractInboundFileSynchronizer` provides a `removeRemoteFileMetadata()` callback for this purpose. In addition there is a `setMetadataStorePrefix()` to be used in the metadata keys. It is recommended to have this prefix be different from the one used in the `MetadataStore`-based `FileListFilter` implementations, when the same `MetadataStore` instance is shared between these components, to avoid entry overriding because both filter and `AbstractInboundFileSynchronizer` use the same local file name for the metadata entry key.

Chapter 20. Pivotal GemFire and Apache Geode Support

Spring Integration provides support for Pivotal GemFire and Apache Geode.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-gemfire</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-gemfire:5.3.0.RC1"
```

GemFire is a distributed data management platform that provides a key-value data grid along with advanced distributed system features, such as event processing, continuous querying, and remote function execution. This guide assumes some familiarity with the commercial Pivotal GemFire or Open Source Apache Geode.

Spring integration provides support for GemFire by implementing inbound adapters for entry and continuous query events, an outbound adapter to write entries to the cache, and message and metadata stores and `GemfireLockRegistry` implementations. Spring integration leverages the Spring Data for Pivotal GemFire project, providing a thin wrapper over its components.

Starting with version 5.1, the Spring Integration GemFire module uses the Spring Data for Apache Geode transitive dependency by default. To switch to the commercial Pivotal GemFire-based Spring Data for Pivotal GemFire, exclude `spring-data-geode` from dependencies and add `spring-data-gemfire`, as the following Maven snippet shows:

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-gemfire</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-geode</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-gemfire</artifactId>
</dependency>
```

To configure the 'int-gfe' namespace, include the following elements within the headers of your XML configuration file:

```
xmlns:int-gfe="http://www.springframework.org/schema/integration/gemfire"
xsi:schemaLocation="http://www.springframework.org/schema/integration/gemfire
    https://www.springframework.org/schema/integration/gemfire/spring-integration-
gemfire.xsd"
```

20.1. Inbound Channel Adapter

The inbound channel adapter produces messages on a channel when triggered by a GemFire `EntryEvent`. GemFire generates events whenever an entry is `CREATED`, `UPDATED`, `DESTROYED`, or `INVALIDATED` in the associated region. The inbound channel adapter lets you filter on a subset of these events. For example, you may want to produce messages only in response to an entry being created. In addition, the inbound channel adapter can evaluate a SpEL expression if, for example, you want your message payload to contain an event property such as the new entry value. The following example shows how to configure an inbound channel adapter with a SpEL language (in the `expression` attribute):

```
<gfe:cache/>
<gfe:replicated-region id="region"/>
<int-gfe:inbound-channel-adapter id="inputChannel" region="region"
    cache-events="CREATED" expression="newValue"/>
```

The preceding configuration creates a GemFire `Cache` and `Region` by using Spring GemFire's 'gfe'

namespace. The `inbound-channel-adapter` element requires a reference to the GemFire region on which the adapter listens for events. Optional attributes include `cache-events`, which can contain a comma-separated list of event types for which a message is produced on the input channel. By default, `CREATED` and `UPDATED` are enabled. If no `channel` attribute is provided, the channel is created from the `id` attribute. This adapter also supports an `error-channel`. The GemFire `EntryEvent` is the `#root` object of the `expression` evaluation. The following example shows an expression that replaces a value for a key:

```
expression="new something.MyEvent(key, oldValue, newValue)"
```

If the `expression` attribute is not provided, the message payload is the GemFire `EntryEvent` itself.

This adapter conforms to Spring Integration conventions.

20.2. Continuous Query Inbound Channel Adapter

The continuous query inbound channel adapter produces messages on a channel when triggered by a GemFire continuous query or `CqEvent` event. In release `1.1`, Spring Data introduced continuous query support, including `ContinuousQueryListenerContainer`, which provides a nice abstraction over the GemFire native API. This adapter requires a reference to a `ContinuousQueryListenerContainer` instance, creates a listener for a given `query`, and executes the query. The continuous query acts as an event source that fires whenever its result set changes state.

GemFire queries are written in OQL and are scoped to the entire cache (not just one region). Additionally, continuous queries require a remote (that is, running in a separate process or remote host) cache server. See the GemFire documentation for more information on implementing continuous queries.

The following configuration creates a GemFire client cache (recall that a remote cache server is required for this implementation and its address is configured as a child element of the pool), a client region, and a `ContinuousQueryListenerContainer` that uses Spring Data:

```
<gfe:client-cache id="client-cache" pool-name="client-pool"/>

<gfe:pool id="client-pool" subscription-enabled="true" >
    <!--configure server or locator here required to address the cache server -->
</gfe:pool>

<gfe:client-region id="test" cache-ref="client-cache" pool-name="client-pool"/>

<gfe:cq-listener-container id="queryListenerContainer" cache="client-cache"
    pool-name="client-pool"/>

<int-gfe:cq-inbound-channel-adapter id="inputChannel"
    cq-listener-container="queryListenerContainer"
    query="select * from /test"/>
```

The continuous query inbound channel adapter requires a `cq-listener-container` attribute, which must contain a reference to the `ContinuousQueryListenerContainer`. Optionally, it accepts an `expression` attribute that uses SpEL to transform the `CqEvent` or extract an individual property as needed. The `cq-inbound-channel-adapter` provides a `query-events` attribute that contains a comma-separated list of event types for which a message is produced on the input channel. The available event types are `CREATED`, `UPDATED`, `DESTROYED`, `REGION_DESTROYED`, and `REGION_INVALIDATED`. By default, `CREATED` and `UPDATED` are enabled. Additional optional attributes include `query-name` (which provides an optional query name), `expression` (which works as described in the preceding section), and `durable` (a boolean value indicating if the query is durable — it is false by default). If you do not provide a `channel`, the channel is created from the `id` attribute. This adapter also supports an `error-channel`.

This adapter conforms to Spring Integration conventions.

20.3. Outbound Channel Adapter

The outbound channel adapter writes cache entries that are mapped from the message payload. In its simplest form, it expects a payload of type `java.util.Map` and puts the map entries into its configured region. The following example shows how to configure an outbound channel adapter:

```
<int-gfe:outbound-channel-adapter id="cacheChannel" region="region"/>
```

Given the preceding configuration, an exception is thrown if the payload is not a `Map`. Additionally, you can configure the outbound channel adapter to create a map of cache entries by using SpEL. The following example shows how to do so:

```
<int-gfe:outbound-channel-adapter id="cacheChannel" region="region">
    <int-gfe:cache-entries>
        <entry key="payload.toUpperCase()" value="payload.toLowerCase()"/>
        <entry key="'thing1'" value="'thing2'"/>
    </int-gfe:cache-entries>
</int-gfe:outbound-channel-adapter>
```

In the preceding configuration, the inner element (`cache-entries`) is semantically equivalent to a Spring 'map' element. The adapter interprets the `key` and `value` attributes as SpEL expressions with the message as the evaluation context. Note that this can contain arbitrary cache entries (not only those derived from the message) and that literal values must be enclosed in single quotes. In the preceding example, if the message sent to `cacheChannel` has a `String` payload with a value `Hello`, two entries (`[HELLO:hello, thing1:thing2]`) are written (either created or updated) in the cache region. This adapter also supports the `order` attribute, which may be useful if it is bound to a `PublishSubscribeChannel`.

20.4. Gemfire Message Store

As described in EIP, a message store lets you persist messages. This can be useful when dealing with components that have a capability to buffer messages (`QueueChannel`, `Aggregator`, `Resequencer`, and others) if reliability is a concern. In Spring Integration, the `MessageStore` strategy interface also provides the foundation for the claim check pattern, which is described in EIP as well.

Spring Integration's Gemfire module provides `GemfireMessageStore`, which is an implementation of both the the `MessageStore` strategy (mainly used by the `QueueChannel` and `ClaimCheck` patterns) and the `MessageGroupStore` strategy (mainly used by the `Aggregator` and `Resequencer` patterns).

The following example configures the cache and region by using the `spring-gemfire` namespace (not to be confused with the `spring-integration-gemfire` namespace):

```
<bean id="gemfireMessageStore" class="o.s.i.gemfire.store.GemfireMessageStore">
    <constructor-arg ref="myRegion"/>
</bean>

<gfe:cache/>

<gfe:replicated-region id="myRegion"/>


<int:channel id="somePersistentQueueChannel">
    <int:queue message-store="gemfireMessageStore"/>
<int:channel>

<int:aggregator input-channel="inputChannel" output-channel="outputChannel"
    message-store="gemfireMessageStore"/>
```

Often, it is desirable for the message store to be maintained in one or more remote cache servers in a client-server configuration. In this case, you should configure a client cache, a client region, and a client pool and inject the region into the `MessageStore`. The following example shows how to do so:

```
<bean id="gemfireMessageStore"
    class="org.springframework.integration.gemfire.store.GemfireMessageStore">
    <constructor-arg ref="myRegion"/>
</bean>

<gfe:client-cache/>

<gfe:client-region id="myRegion" shortcut="PROXY" pool-name="messageStorePool"/>

<gfe:pool id="messageStorePool">
    <gfe:server host="localhost" port="40404" />
</gfe:pool>
```

Note that the `pool` element is configured with the address of a cache server (you can substitute a locator here). The region is configured as a 'PROXY' so that no data is stored locally. The region's `id` corresponds to a region with the same name in the cache server.

Starting with version 4.3.12, the `GemfireMessageStore` supports the key `prefix` option to allow distinguishing between instances of the store on the same GemFire region.

20.5. Gemfire Lock Registry

Starting with version 4.0, the `GemfireLockRegistry` is available. Certain components (for example, the aggregator and the resequencer) use a lock obtained from a `LockRegistry` instance to ensure

that only one thread is manipulating a group at any given time. The `DefaultLockRegistry` performs this function within a single component. You can now configure an external lock registry on these components. When you use a shared `MessageGroupStore` with the `GemfireLockRegistry`, it can provide this functionality across multiple application instances, such that only one instance can manipulate the group at a time.

One of the `GemfireLockRegistry` constructors requires a `Region` as an argument. It is used to obtain a `Lock` from the `getDistributedLock()` method. This operation requires `GLOBAL` scope for the `Region`. Another constructor requires a `Cache`, and the `Region` is created with `GLOBAL` scope and with the name, `LockRegistry`.

20.6. Gemfire Metadata Store

Version 4.0 introduced a new Gemfire-based `MetadataStore` (Metadata Store) implementation. You can use the `GemfireMetadataStore` to maintain metadata state across application restarts. This new `MetadataStore` implementation can be used with adapters such as:

- Feed Inbound Channel Adapter
- Reading Files
- FTP Inbound Channel Adapter
- SFTP Inbound Channel Adapter

To get these adapters to use the new `GemfireMetadataStore`, declare a Spring bean with a bean name of `metadataStore`. The feed inbound channel adapter automatically picks up and use the declared `GemfireMetadataStore`.

The `GemfireMetadataStore` also implements `ConcurrentMetadataStore`, letting it be reliably shared across multiple application instances, where only one instance can store or modify a key's value. These methods give various levels of concurrency guarantees based on the scope and data policy of the region. They are implemented in the peer cache and client-server cache but are disallowed in peer regions that have `NORMAL` or `EMPTY` data policies.

Since version 5.0, the `GemfireMetadataStore` also implements `ListenableMetadataStore`, which lets you listen to cache events by providing `MetadataStoreListener` instances to the store, as the following example shows:

```
GemfireMetadataStore metadataStore = new GemfireMetadataStore(cache);
metadataStore.addListener(new MetadataStoreListenerAdapter() {

    @Override
    public void onAdd(String key, String value) {
         ...
    }

});
```

Chapter 21. HTTP Support

Spring Integration's HTTP support allows for the running of HTTP requests and the processing of inbound HTTP requests. The HTTP support consists of the following gateway implementations: `HttpInboundEndpoint` and `HttpRequestExecutingMessageHandler`. See also WebFlux Support.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-http</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-http:5.3.0.RC1"
```

The `javax.servlet:javax.servlet-api` dependency must be provided on the target Servlet container.

21.1. Http Inbound Components

To receive messages over HTTP, you need to use an HTTP inbound channel adapter or an HTTP inbound gateway. To support the HTTP inbound adapters, they need to be deployed within a servlet container such as Apache Tomcat or Jetty. The easiest way to do this is to use Spring's `HttpRequestHandlerServlet`, by providing the following servlet definition in the `web.xml` file:

```
<servlet>
    <servlet-name>inboundGateway</servlet-name>
    <servlet-class>o.s.web.context.support.HttpRequestHandlerServlet</servlet-
class>
</servlet>
```

Notice that the servlet name matches the bean name. For more information on using the `HttpRequestHandlerServlet`, see Remoting and web services using Spring, which is part of the Spring Framework Reference documentation.

If you are running within a Spring MVC application, then the aforementioned explicit servlet definition is not necessary. In that case, the bean name for your gateway can be matched against the URL path as you would for a Spring MVC Controller bean. For more information, see Web MVC framework, which is part of the Spring Framework Reference documentation.

For a sample application and the corresponding configuration, see the Spring Integration Samples repository. It contains the HTTP sample application, which demonstrates Spring Integration's HTTP support.

The following example bean defines an HTTP inbound endpoint:

```
<bean id="httpInbound"
  class=
"org.springframework.integration.http.inbound.HttpRequestHandlingMessagingGateway"
>
  <property name="requestChannel" ref="httpRequestChannel" />
  <property name="replyChannel" ref="httpReplyChannel" />
</bean>
```

The `HttpRequestHandlingMessagingGateway` accepts a list of `HttpMessageConverter` instances or else relies on a default list. The converters allow customization of the mapping from `HttpServletRequest` to `Message`. The default converters encapsulate simple strategies, which (for example) create a `String` message for a `POST` request where the content type starts with `text`. See the Javadoc for full details. An additional flag (`mergeWithDefaultConverters`) can be set along with the list of custom `HttpMessageConverter` to add the default converters after the custom converters. By default, this flag is set to `false`, meaning that the custom converters replace the default list.

The message conversion process uses the (optional) `requestPayloadType` property and the incoming `Content-Type` header. Starting with version 4.3, if a request has no content type header, `application/octet-stream` is assumed, as recommended by `RFC 2616`. Previously, the body of such messages was ignored.

Spring Integration 2.0 implemented multipart file support. If the request has been wrapped as a `MultipartHttpServletRequest`, when you use the default converters, that request is converted to a `Message` payload that is a `MultiValueMap` containing values that may be byte arrays, strings, or instances of Spring's `MultipartFile`, depending on the content type of the individual parts.

The HTTP inbound endpoint locates a `MultipartResolver` in the context if one has a bean name of `multipartResolver` (the same name expected by Spring's `DispatcherServlet`). If it does locate that bean, the support for multipart files is enabled on the inbound request mapper. Otherwise, it fails when it tries to map a multipart file request to a Spring Integration `Message`. For more on Spring's support for `MultipartResolver`, see the Spring Reference Manual.

If you wish to proxy a `multipart/form-data` to another server, it may be better to keep it in raw form. To handle this situation, do not add the `multipartResolver` bean to the context. Configure the endpoint to expect a `byte[]` request, customize the message converters to include a `ByteArrayHttpMessageConverter`, and disable the default multipart converter. You may need some other converters for the replies. The following example shows such an arrangement:

```
<int-http:inbound-gateway
                  channel="receiveChannel"
                  path="/inboundAdapter.htm"
                  request-payload-type="byte[]"
                  message-converters="converters"
                  merge-with-default-converters="false"
                  supported-methods="POST" />

<util:list id="converters">
    <beans:bean class=
"org.springframework.http.converter.ByteArrayHttpMessageConverter" />
    <beans:bean class=
"org.springframework.http.converter.StringHttpMessageConverter" />
    <beans:bean class=
"org.springframework.http.converter.json.MappingJackson2HttpMessageConv
erter" />
</util:list>
```

When you send a response to the client, you have a number of ways to customize the behavior of the gateway. By default, the gateway acknowledges that the request was received by sending a `200` status code back. It is possible to customize this response by providing a 'viewName' to be resolved by the Spring MVC `ViewResolver`. If the gateway should expect a reply to the `Message`, you can set the `expectReply` flag (constructor argument) to cause the gateway to wait for a reply `Message` before creating an HTTP response. The following example configures a gateway to serve as a Spring MVC Controller with a view name:

```
<bean id="httpInbound"
  class="org.springframework.integration.http.inbound.HttpRequestHandlingController">
  <constructor-arg value="true" /> <!-- indicates that a reply is expected -->
  <property name="requestChannel" ref="httpRequestChannel" />
  <property name="replyChannel" ref="httpReplyChannel" />
  <property name="viewName" value="jsonView" />
  <property name="supportedMethodNames" >
    <list>
      <value>GET</value>
      <value>DELETE</value>
    </list>
  </property>
</bean>
```

Because of the `constructor-arg` value of `true`, it waits for a reply. The preceding example also shows how to customize the HTTP methods accepted by the gateway, which are `POST` and `GET` by default.

The reply message is available in the model map. By default, the key for that map entry is 'reply', but you can override this default by setting the 'replyKey' property on the endpoint's configuration.

21.1.1. Payload Validation

Starting with version 5.2, the HTTP inbound endpoints can be supplied with a `Validator` to check a payload before sending into the channel. This payload is already a result of conversion and extraction after `payloadExpression` to narrow a validation scope in regards to the valuable data. The validation failure handling is fully the same what we have in Spring MVC Error Handling.

21.2. HTTP Outbound Components

This section describes Spring Integration's HTTP outbound components.

21.2.1. Using `HttpRequestExecutingMessageHandler`

To configure the `HttpRequestExecutingMessageHandler`, write a bean definition similar to the following:

```
<bean id="httpOutbound"
  class=
"org.springframework.integration.http.outbound.HttpRequestExecutingMessageHandler"
>
  <constructor-arg value="http://localhost:8080/example" />
  <property name="outputChannel" ref="responseChannel" />
</bean>
```

This bean definition runs HTTP requests by delegating to a `RestTemplate`. That template, in turn, delegates to a list of `HttpMessageConverter` instances to generate the HTTP request body from the `Message` payload. You can configure those converters as well as the `ClientHttpRequestFactory` instance to use, as the following example shows:

```
<bean id="httpOutbound"
  class=
"org.springframework.integration.http.outbound.HttpRequestExecutingMessageHandler"
>
  <constructor-arg value="http://localhost:8080/example" />
  <property name="outputChannel" ref="responseChannel" />
  <property name="messageConverters" ref="messageConverterList" />
  <property name="requestFactory" ref="customRequestFactory" />
</bean>
```

By default, the HTTP request is generated by using an instance of `SimpleClientHttpRequestFactory`, which uses the JDK `HttpURLConnection`. Use of the Apache Commons HTTP Client is also supported through `CommonsClientHttpRequestFactory`, which you can inject (as shown earlier).

For the outbound gateway, the reply message produced by the gateway contains all the message headers that are present in the request message.

21.2.2. Using Cookies

Basic cookie support is provided by the `transfer-cookies` attribute on the outbound gateway. When set to `true` (the default is `false`), a `Set-Cookie` header received from the server in a response is converted to `Cookie` in the reply message. This header is then used on subsequent sends. This enables simple stateful interactions, such as the following:

`···→logonGateway→···→doWorkGateway→···→logoffGateway→···`

If `transfer-cookies` is `false`, any `Set-Cookie` header received remains as `Set-Cookie` in the reply message and is dropped on subsequent sends.

Empty Response Bodies

HTTP is a request-response protocol. However, the response may not have a body, only headers. In this case, the `HttpRequestExecutingMessageHandler` produces a reply `Message` with the payload being an `org.springframework.http.ResponseEntity`, regardless of any provided `expected-response-type`. According to the HTTP RFC Status Code Definitions, there are many statuses that mandate that a response must not contain a message-body (for example, `204 No Content`). There are also cases where calls to the same URL might or might not return a response body. For example, the first request to an HTTP resource returns content, but the second does not (returning a `304 Not Modified`). In all cases, however, the `http_statusCode` message header is populated. This can be used in some routing logic after the HTTP outbound gateway. You could also use a`<payload-type-router/>` to route messages with a `ResponseEntity` to a different flow than that used for responses with a body.

expected-response-type

Further to the preceding note about empty response bodies, if a response does contain a body, you must provide an appropriate `expected-response-type` attribute or, again, you receive a `ResponseEntity` with no body. The `expected-response-type` must be compatible with the (configured or default) `HttpMessageConverter` instances and the `Content-Type` header in the response. This can be an abstract class or even an interface (such as `java.io.Serializable` when you use Java serialization and `Content-Type: application/x-java-serialized-object`).

21.3. HTTP Namespace Support

Spring Integration provides an `http` namespace and the corresponding schema definition. To include it in your configuration, provide the following namespace declaration in your application

context configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-http="http://www.springframework.org/schema/integration/http"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    https://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/http
    https://www.springframework.org/schema/integration/http/spring-integration-
http.xsd">
    ...
</beans>
```

21.3.1. Inbound

The XML namespace provides two components for handling HTTP inbound requests: `inbound-channel-adapter` and `inbound-gateway`. In order to process requests without returning a dedicated response, use the `inbound-channel-adapter`. The following example shows how to configure one:

```
<int-http:inbound-channel-adapter id="httpChannelAdapter" channel="requests"
    supported-methods="PUT, DELETE"/>
```

To process requests that do expect a response, use an `inbound-gateway`. The following example shows how to configure one:

```
<int-http:inbound-gateway id="inboundGateway"
    request-channel="requests"
    reply-channel="responses"/>
```

21.3.2. Request Mapping Support

Spring Integration 3.0 improved the REST support by introducing the `IntegrationRequestMappingHandlerMapping`. The implementation relies on the enhanced REST support provided by Spring Framework 3.1 or higher.

The parsing of the HTTP inbound gateway or the HTTP inbound channel adapter registers an

integrationRequestMappingHandlerMapping bean of type IntegrationRequestMappingHandlerMapping, in case one is not yet registered. This particular implementation of the HandlerMapping delegates its logic to RequestMappingInfoHandlerMapping. The implementation provides functionality similar to the org.springframework.web.bind.annotation.RequestMapping annotation in Spring MVC.

For more information, see Mapping Requests With @RequestMapping.

For this purpose, Spring Integration 3.0 introduces the <request-mapping> element. You can add this optional element to <http:inbound-channel-adapter> and <http:inbound-gateway>. It works in conjunction with the path and supported-methods attributes. The following example shows how to configure it on an inbound gateway:

```
<inbound-gateway id="inboundController"
    request-channel="requests"
    reply-channel="responses"
    path="/foo/{fooId}"
    supported-methods="GET"
    view-name="foo"
    error-code="oops">
   <request-mapping headers="User-Agent"
     params="myParam=myValue"
     consumes="application/json"
     produces="!text/plain"/>
</inbound-gateway>
```

Based on the preceding configuration, the namespace parser creates an instance of the IntegrationRequestMappingHandlerMapping (if none exists) and an HttpRequestHandlingController bean and associates with it an instance of RequestMapping. This RequestMapping instance is, in turn, converted to the Spring MVC RequestMappingInfo.

The <request-mapping> element provides the following attributes:

- headers
- params
- consumes
- produces

With the path and supported-methods attributes of the <http:inbound-channel-adapter> or the <http:inbound-gateway>, <request-mapping> attributes translate directly into the respective options provided by the org.springframework.web.bind.annotation.RequestMapping annotation in Spring MVC.

The <request-mapping> element lets you configure several Spring Integration HTTP inbound endpoints to the same path (or even the same supported-methods) and lets you provide different downstream message flows based on incoming HTTP requests.

Alternatively, you can also declare only one HTTP inbound endpoint and apply routing and filtering logic within the Spring Integration flow to achieve the same result. This lets you get the `Message` into the flow as early as possibly. The following example shows how to do so:

```
<int-http:inbound-gateway request-channel="httpMethodRouter"
    supported-methods="GET,DELETE"
    path="/process/{entId}"
    payload-expression="#pathVariables.entId"/>

<int:router input-channel="httpMethodRouter" expression=
"headers.http_requestMethod">
    <int:mapping value="GET" channel="in1"/>
    <int:mapping value="DELETE" channel="in2"/>
</int:router>

<int:service-activator input-channel="in1" ref="service" method="getEntity"/>

<int:service-activator input-channel="in2" ref="service" method="delete"/>
```

For more information regarding handler mappings, see the Spring Framework Web Servlet documentation or the Spring Framework Web Reactive documentation.

21.3.3. Cross-origin Resource Sharing (CORS) Support

Starting with version 4.2, you can configure the `<http:inbound-channel-adapter>` and `<http:inbound-gateway>` with a `<cross-origin>` element. It represents the same options as Spring MVC's `@CrossOrigin` for `@Controller` annotations and allows the configuration of cross-origin resource sharing (CORS) for Spring Integration HTTP endpoints:

- `origin`: List of allowed origins. The `*` means that all origins are allowed. These values are placed in the `Access-Control-Allow-Origin` header of both the pre-flight and actual responses. The default value is `*`.
- `allowed-headers`: Indicates which request headers can be used during the actual request. The `*` means that all headers requested by the client are allowed. This property controls the value of the pre-flight response's `Access-Control-Allow-Headers` header. The default value is `*`.
- `exposed-headers`: List of response headers that the user-agent lets the client access. This property controls the value of the actual response's `Access-Control-Expose-Headers` header.
- `method`: The HTTP request methods to allow: `GET`, `POST`, `HEAD`, `OPTIONS`, `PUT`, `PATCH`, `DELETE`, `TRACE`. Methods specified here overrides those in `supported-methods`.
- `allow-credentials`: Set to `true` if the the browser should include any cookies associated to the domain of the request or `false` if it should not. An empty string ("") means undefined. If `true`, the pre-flight response includes the `Access-Control-Allow-Credentials=true` header. The default value is `true`.
- `max-age`: Controls the cache duration for pre-flight responses. Setting this to a reasonable value can reduce the number of pre-flight request-response interactions required by the browser.

This property controls the value of the `Access-Control-Max-Age` header in the pre-flight response. A value of `-1` means undefined. The default value is 1800 seconds (30 minutes).

The CORS Java Configuration is represented by the `org.springframework.integration.http.inbound.CrossOrigin` class, instances of which can be injected into the `HttpRequestHandlingEndpointSupport` beans.

21.3.4. Response Status Code

Starting with version 4.1, you can configure the `<http:inbound-channel-adapter>` with a `status-code-expression` to override the default `200 OK` status. The expression must return an object that can be converted to an `org.springframework.http.HttpStatus` enum value. The `evaluationContext` has a `BeanResolver` and, starting with version 5.1, is supplied with the `RequestEntity<?>` as root object. An example might be to resolve, at runtime, some scoped bean that returns a status code value. However, most likely, it is set to a fixed value such as `status-code=expression="204"` (No Content), or `status-code-expression="T(org.springframework.http.HttpStatus).NO_CONTENT"`. By default, `status-code-expression` is null, meaning that the normal '200 OK' response status is returned. Using the `RequestEntity<?>` as root object, the status code can be conditional e.g. on the request method, some header, URI content or even request body. The following example shows how to set the status code to `ACCEPTED`:

```
<http:inbound-channel-adapter id="inboundController"
       channel="requests" view-name="foo" error-code="oops"
       status-code-expression="T(org.springframework.http.HttpStatus).ACCEPTED">
   <request-mapping headers="BAR"/>
</http:inbound-channel-adapter>
```

The `<http:inbound-gateway>` resolves the 'status code' from the `http_statusCode` header of the reply `Message`. Starting with version 4.2, the default response status code when no reply is received within the `reply-timeout` is `500 Internal Server Error`. There are two ways to modify this behavior:

- Add a `reply-timeout-status-code-expression`. This has the same semantics as the `status-code-expression` on the inbound adapter.
- Add an `error-channel` and return an appropriate message with an HTTP status code header, as the following example shows:

```
<int:chain input-channel="errors">
    <int:header-enricher>
        <int:header name="http_statusCode" value="504" />
    </int:header-enricher>
    <int:transformer expression="payload.failedMessage" />
</int:chain>
```

The payload of the `ErrorMessage` is a `MessageTimeoutException`. It must be transformed to something that can be converted by the gateway, such as a `String`. A good candidate is the exception's message property, which is the value used when you use the `expression` technique.

If the error flow times out after a main flow timeout, `500 Internal Server Error` is returned, or, if the `reply-timeout-status-code-expression` is present, it is evaluated.

Previously, the default status code for a timeout was `200 OK`. To restore that behavior, set `reply-timeout-status-code-expression="200"`.

21.3.5. URI Template Variables and Expressions

By using the `path` attribute in conjunction with the `payload-expression` attribute and the `header` element, you have a high degree of flexibility for mapping inbound request data.

In the following example configuration, an inbound channel adapter is configured to accept requests using the following URI:

```
/first-name/{firstName}/last-name/{lastName}
```

When you use the `payload-expression` attribute, the `Mark` URI template variable maps to be the `Message` payload, while the `Fisher` URI template variable maps to the `lname` message header, as defined in the following example:

```
<int-http:inbound-channel-adapter id="inboundAdapterWithExpressions"
    path="/first-name/{firstName}/last-name/{lastName}"
    channel="requests"
    payload-expression="#pathVariables.firstName">
    <int-http:header name="lname" expression="#pathVariables.lastName"/>
</int-http:inbound-channel-adapter>
```

For more information about URI template variables, see uri template patterns in the Spring Reference Manual.

Since Spring Integration 3.0, in addition to the existing `#pathVariables` and `#requestParams` variables being available in payload and header expressions, we added other useful expression variables:

- `#requestParams`: The `MultiValueMap` from the `ServletRequest parameterMap`.
- `#pathVariables`: The `Map` from URI Template placeholders and their values.
- `#matrixVariables`: The `Map` of `MultiValueMap` according to the Spring MVC Specification. Note that `#matrixVariables` requires Spring MVC 3.2 or higher.
- `#requestAttributes`: The `org.springframework.web.context.request.RequestAttributes` associated with the current request.

- `#requestHeaders`: The `org.springframework.http.HttpHeaders` object from the current request.
- `#cookies`: The `Map<String, Cookie>` of `javax.servlet.http.Cookie` instances from the current request.

Note that all these values (and others) can be accessed within expressions in the downstream message flow through the `ThreadLocal org.springframework.web.context.request.RequestAttributes` variable, if that message flow is single-threaded and lives within the request thread. The following example configures a transformer that uses an `expression` attribute:

```
<int-:transformer
    expression="T(org.springframework.web.context.request.RequestContextHolder).
                requestAttributes.request.queryString"/>
```

21.3.6. Outbound

To configure the outbound gateway, you can use the namespace support. The following code snippet shows the available configuration options for an outbound HTTP gateway:

```
<int-http:outbound-gateway id="example"
    request-channel="requests"
    url="http://localhost/test"
    http-method="POST"
    extract-request-payload="false"
    expected-response-type="java.lang.String"
    charset="UTF-8"
    request-factory="requestFactory"
    reply-timeout="1234"
    reply-channel="replies"/>
```

Most importantly, notice that the 'http-method' and 'expected-response-type' attributes are provided. Those are two of the most commonly configured values. The default `http-method` is `POST`, and the default response type is null. With a null response type, the payload of the reply `Message` contains the `ResponseEntity`, as long as its HTTP status is a success (non-successful status codes throw exceptions). If you expect a different type, such as a `String`, provide that as a fully-qualified class name (`java.lang.String` in the preceding example). See also the note about empty response bodies in HTTP Outbound Components.

Beginning with Spring Integration 2.1, the `request-timeout` attribute of the HTTP outbound gateway was renamed to `reply-timeout` to better reflect its intent.

Since Spring Integration 2.2, Java serialization over HTTP is no longer enabled by default. Previously, when setting the `expected-response-type` attribute to a `Serializable` object, the `Accept` header was not properly set up. Since Spring Integration 2.2, the `SerializingHttpMessageConverter` has now been updated to set the `Accept` header to `application/x-java-serialized-object`.

However, because this could cause incompatibility with existing applications, it was decided to no longer automatically add this converter to the HTTP endpoints. If you wish to use Java serialization, you can add the `SerializingHttpMessageConverter` to the appropriate endpoints, by using the `message-converters` attribute (when you use XML configuration) or by using the `setMessageConverters()` method (in Java configuration). Alternatively, you may wish to consider using JSON instead, which is enabled by having the Jackson library on the classpath.

Beginning with Spring Integration 2.2, you can also determine the HTTP method dynamically by using SpEL and the `http-method-expression` attribute. Note that this attribute is mutually exclusive with `http-method`. You can also use the `expected-response-type-expression` attribute instead of `expected-response-type` and provide any valid SpEL expression that determines the type of the response. The following configuration example uses `expected-response-type-expression`:

```
<int-http:outbound-gateway id="example"
    request-channel="requests"
    url="http://localhost/test"
    http-method-expression="headers.httpMethod"
    extract-request-payload="false"
    expected-response-type-expression="payload"
    charset="UTF-8"
    request-factory="requestFactory"
    reply-timeout="1234"
    reply-channel="replies"/>
```

If your outbound adapter is to be used in a unidirectional way, you can use an `outbound-channel-adapter` instead. This means that a successful response executes without sending any messages to a reply channel. In the case of any non-successful response status code, it throws an exception. The configuration looks very similar to the gateway, as the following example shows:

```
<int-http:outbound-channel-adapter id="example"
    url="http://localhost/example"
    http-method="GET"
    channel="requests"
    charset="UTF-8"
    extract-payload="false"
    expected-response-type="java.lang.String"
    request-factory="someRequestFactory"
    order="3"
    auto-startup="false"/>
```

To specify the URL, you can use either the 'url' attribute or the 'url-expression' attribute. The 'url' attribute takes a simple string (with placeholders for URI variables, as described below). The 'url-expression' is a SpEL expression, with the `Message` as the root object, which enables dynamic urls. The URL that results from the expression evaluation can still have placeholders for URI variables.

In previous releases, some users used the place holders to replace the entire URL with a URI variable. Changes in Spring 3.1 can cause some issues with escaped characters, such as '?'. For this reason, we recommend that, if you wish to generate the URL entirely at runtime, you use the 'url-expression' attribute.

21.3.7. Mapping URI Variables

If your URL contains URI variables, you can map them by using the `uri-variable` element. This element is available for the HTTP outbound gateway and the HTTP outbound channel adapter. The following example maps the `zipCode` URI variable to an expression:

```
<int-http:outbound-gateway id="trafficGateway"
    url="https://local.yahooapis.com/trafficData?appid=YdnDemo&amp;zip={zipCode}"
    request-channel="trafficChannel"
    http-method="GET"
    expected-response-type="java.lang.String">
    <int-http:uri-variable name="zipCode" expression="payload.getZip()"/>
</int-http:outbound-gateway>
```

The `uri-variable` element defines two attributes: `name` and `expression`. The `name` attribute identifies the name of the URI variable, while the `expression` attribute is used to set the actual value. By using the `expression` attribute, you can leverage the full power of the Spring Expression Language (SpEL), which gives you full dynamic access to the message payload and the message headers. For example, in the preceding configuration, the `getZip()` method is invoked on the payload object of the `Message` and the result of that method is used as the value of the URI variable named 'zipCode'.

Since Spring Integration 3.0, HTTP outbound endpoints support the `uri-variables-expression`

attribute to specify an `expression` that should be evaluated, resulting in a `Map` of all URI variable placeholders within the URL template. It provides a mechanism whereby you can use different variable expressions, based on the outbound message. This attribute is mutually exclusive with the `<uri-variable/>` element. The following example shows how to use the `uri-variables-expression` attribute:

```
<int-http:outbound-gateway
     url="https://foo.host/{foo}/bars/{bar}"
     request-channel="trafficChannel"
     http-method="GET"
     uri-variables-expression="@uriVariablesBean.populate(payload)"
     expected-response-type="java.lang.String"/>
```

`uriVariablesBean` might be defined as follows:

```
public class UriVariablesBean {
    private static final ExpressionParser EXPRESSION_PARSER = new
SpelExpressionParser();

    public Map<String, ?> populate(Object payload) {
        Map<String, Object> variables = new HashMap<String, Object>();
        if (payload instanceOf String.class)) {
            variables.put("foo", "foo"));
        }
        else {
            variables.put("foo", EXPRESSION_PARSER.parseExpression("headers.bar")
);
        }
        return variables;
    }

}
```

The `uri-variables-expression` must evaluate to a `Map`. The values of the `Map` must be instances of `String` or `Expression`. This `Map` is provided to an `ExpressionEvalMap` for further resolution of URI variable placeholders by using those expressions in the context of the outbound `Message`.

IMPORTANT

The `uriVariablesExpression` property provides a very powerful mechanism for evaluating URI variables. We anticipate that people mostly use simple expressions, such as the preceding example. However, you can also configure something such as `"@uriVariablesBean.populate(#root)"` with an expression in the returned map being `variables.put("thing1", EXPRESSION_PARSER.parseExpression(message.getHeaders().get("thing2", String.class)));`, where the expression is dynamically provided in the message header named `thing2`. Since the header may come from an untrusted source, the HTTP outbound endpoints use `SimpleEvaluationContext` when evaluating these expressions. The `SimpleEvaluationContext` uses only a subset of SpEL features. If you trust your message sources and wish to use the restricted SpEL constructs, set the `trustedSpel` property of the outbound endpoint to `true`.

You can achieve scenarios that need to supply a dynamic set of URI variables on a per-message basis by using a custom `url-expression` and some utilities for building and encoding URL parameters. The following example shows how to do so:

```
url-expression="T(org.springframework.web.util.UriComponentsBuilder)
                            .fromHttpUrl('https://HOST:PORT/PATH')
                            .queryParams(payload)
                            .build()
                            .toUri()"
```

The `queryParams()` method expects a `MultiValueMap<String, String>` as an argument, so you can build a real set of URL query parameters in advance, before performing the request.

The whole `queryString` can also be presented as a `uri-variable`, as the following example shows:

```
<int-http:outbound-gateway id="proxyGateway" request-channel="testChannel"
              url="http://testServer/test?{queryString}">
    <int-http:uri-variable name="queryString" expression="'a=A&amp;b=B'"/>
</int-http:outbound-gateway>
```

In this case, you must manually provide the URL encoding. For example, you can use the `org.apache.http.client.utils.URLEncodedUtils#format()` for this purpose. As mentioned earlier, a manually built `MultiValueMap<String, String>` can be converted to the the `List<NameValuePair> format()` method argument by using the following Java Streams snippet:

```
List<NameValuePair> nameValuePairs =
    params.entrySet()
            .stream()
            .flatMap(e -> e
                    .getValue()
                    .stream()
                    .map(v -> new BasicNameValuePair(e.getKey(), v)))
            .collect(Collectors.toList());
```

21.3.8. Controlling URI Encoding

By default, the URL string is encoded (see `UriComponentsBuilder`) to the URI object before sending the request. In some scenarios with a non-standard URI (such as the RabbitMQ REST API), it is undesirable to perform the encoding. The `<http:outbound-gateway/>` and `<http:outbound-channel-adapter/>` provide an `encoding-mode` attribute. To disable encoding the URL, set this attribute to `NONE` (by default, it is `TEMPLATE_AND_VALUES`). If you wish to partially encode some of the URL, use an `expression` within a `<uri-variable/>`, as the following example shows:

```
<http:outbound-gateway url="https://somehost/%2f/fooApps?bar={param}" encoding-
mode="NONE">
          <http:uri-variable name="param"
            expression="T(org.apache.commons.httpclient.util.URIUtil)
                                          .encodeWithinQuery('Hello World!')"/>
</http:outbound-gateway>
```

With Java DSL this option can be controlled by the `BaseHttpMessageHandlerSpec.encodingMode()` option. The same configuration applies for similar outbound components in the WebFlux module and Web Services module. For much sophisticated scenarios it is recommended to configure an `UriTemplateHandler` on the externally provided `RestTemplate`; or in case of WebFlux - `WebClient` with it `UriBuilderFactory`.

21.4. Configuring HTTP Endpoints with Java

The following example shows how to configure an inbound gateway with Java:

Example 6. Inbound Gateway Using Java Configuration

```
@Bean
public HttpRequestHandlingMessagingGateway inbound() {
    HttpRequestHandlingMessagingGateway gateway =
        new HttpRequestHandlingMessagingGateway(true);
    gateway.setRequestMapping(mapping());
    gateway.setRequestPayloadType(String.class);
    gateway.setRequestChannelName("httpRequest");
    return gateway;
}

@Bean
public RequestMapping mapping() {
    RequestMapping requestMapping = new RequestMapping();
    requestMapping.setPathPatterns("/foo");
    requestMapping.setMethods(HttpMethod.POST);
    return requestMapping;
}
```

The following example shows how to configure an inbound gateway with the Java DSL:

Example 7. Inbound Gateway Using the Java DSL

```
@Bean
public IntegrationFlow inbound() {
    return IntegrationFlows.from(Http.inboundGateway("/foo")
            .requestMapping(m -> m.methods(HttpMethod.POST))
            .requestPayloadType(String.class))
        .channel("httpRequest")
        .get();
}
```

The following example shows how to configure an outbound gateway with Java:

Example 8. Outbound Gateway Using Java Configuration

```
@ServiceActivator(inputChannel = "httpOutRequest")
@Bean
public HttpRequestExecutingMessageHandler outbound() {
    HttpRequestExecutingMessageHandler handler =
        new HttpRequestExecutingMessageHandler("http://localhost:8080/foo");
    handler.setHttpMethod(HttpMethod.POST);
    handler.setExpectedResponseType(String.class);
    return handler;
}
```

The following example shows how to configure an outbound gateway with the Java DSL:

Example 9. Outbound Gateway Using the Java DSL

```
@Bean
public IntegrationFlow outbound() {
    return IntegrationFlows.from("httpOutRequest")
        .handle(Http.outboundGateway("http://localhost:8080/foo")
            .httpMethod(HttpMethod.POST)
            .expectedResponseType(String.class))
        .get();
}
```

21.5. Timeout Handling

In the context of HTTP components, there are two timing areas that have to be considered:

- Timeouts when interacting with Spring Integration Channels
- Timeouts when interacting with a remote HTTP server

The components interact with message channels, for which timeouts can be specified. For example, an HTTP Inbound Gateway forwards messages received from connected HTTP Clients to a message channel (which uses a request timeout) and consequently the HTTP Inbound Gateway receives a reply message from the reply channel (which uses a reply timeout) that is used to generate the HTTP Response. The following illustration offers a visual explanation:

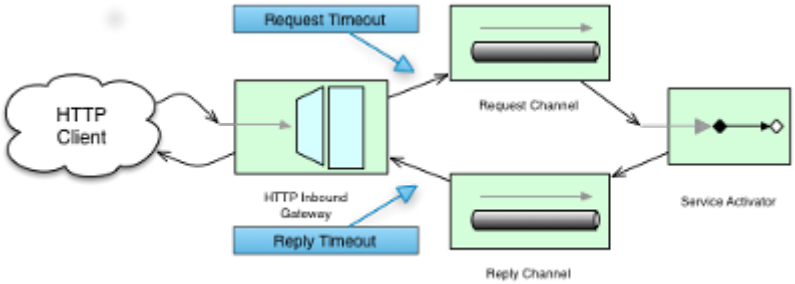


Figure 8. How timeout settings apply to an HTTP Inbound Gateway

For outbound endpoints, we need to consider how timing works while interacting with the remote server. The following image shows this scenario:

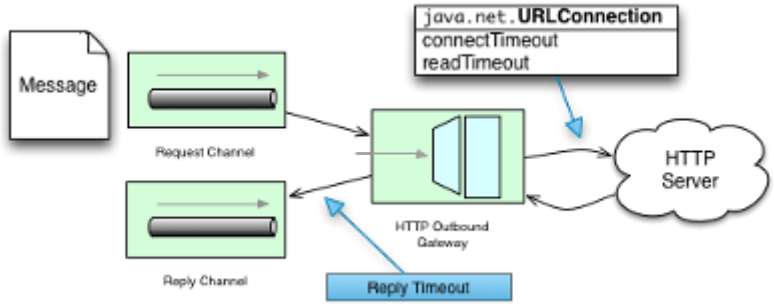


Figure 9. How timeout settings apply to an HTTP Outbound Gateway

You may want to configure the HTTP related timeout behavior, when making active HTTP requests by using the HTTP outbound gateway or the HTTP outbound channel adapter. In those instances, these two components use Spring's `RestTemplate` support to execute HTTP requests.

To configure timeouts for the HTTP outbound gateway and the HTTP outbound channel adapter, you can either reference a `RestTemplate` bean directly (by using the `rest-template` attribute) or you can provide a reference to a `ClientHttpRequestFactory` bean (by using the `request-factory` attribute). Spring provides the following implementations of the `ClientHttpRequestFactory` interface:

- `SimpleClientHttpRequestFactory`: Uses standard J2SE facilities for making HTTP Requests
- `HttpComponentsClientHttpRequestFactory`: Uses Apache HttpComponents HttpClient (since Spring 3.1)

If you do not explicitly configure the `request-factory` or `rest-template` attribute, a default `RestTemplate` (which uses a `SimpleClientHttpRequestFactory`) is instantiated.

With some JVM implementations, the handling of timeouts by the `URLConnection` class may not be consistent.

For example, from the Java™ Platform, Standard Edition 6 API Specification on `setConnectTimeout`:

> Some non-standard implementation of this method may ignore the specified timeout. To see the connect timeout set, please call getConnectTimeout().

If you have specific needs, you should test your timeouts. Consider using the `HttpComponentsClientHttpRequestFactory`, which, in turn, uses Apache HttpComponents HttpClient rather than relying on implementations provided by a JVM.

When you use the Apache HttpComponents HttpClient with a pooling connection manager, you should be aware that, by default, the connection manager creates no more than two concurrent connections per given route and no more than 20 connections in total. For many real-world applications, these limits may prove to be too constraining. See the Apache documentation for information about configuring this important component.

The following example configures an HTTP outbound gateway by using a `SimpleClientHttpRequestFactory` that is configured with connect and read timeouts of 5 seconds, respectively:

```
<int-http:outbound-gateway url=
"https://samples.openweathermap.org/data/2.5/weather?q={city}"
                           http-method="GET"
                           expected-response-type="java.lang.String"
                           request-factory="requestFactory"
                           request-channel="requestChannel"
                           reply-channel="replyChannel">
    <int-http:uri-variable name="city" expression="payload"/>
</int-http:outbound-gateway>

<bean id="requestFactory"
      class="org.springframework.http.client.SimpleClientHttpRequestFactory">
    <property name="connectTimeout" value="5000"/>
    <property name="readTimeout"    value="5000"/>
</bean>
```

HTTP Outbound Gateway

For the *HTTP Outbound Gateway*, the XML Schema defines only the *reply-timeout*. The *reply-timeout* maps to the *sendTimeout* property of the

org.springframework.integration.http.outbound.HttpRequestExecutingMessageHandler class. More precisely, the property is set on the extended `AbstractReplyProducingMessageHandler` class, which ultimately sets the property on the `MessagingTemplate`.

The value of the *sendTimeout* property defaults to "-1" and will be applied to the connected `MessageChannel`. This means, that depending on the implementation, the Message Channel's *send* method may block indefinitely. Furthermore, the *sendTimeout* property is only used, when the actual MessageChannel implementation has a blocking send (such as 'full' bounded QueueChannel).

21.5.1. HTTP Inbound Gateway

For the HTTP inbound gateway, the XML Schema defines the `request-timeout` attribute, which is used to set the `requestTimeout` property on the `HttpRequestHandlingMessagingGateway` class (on the extended `MessagingGatewaySupport` class). You can also use the `reply-timeout` attribute to map to the `replyTimeout` property on the same class.

The default for both timeout properties is `1000ms` (one thousand milliseconds or one second). Ultimately, the `request-timeout` property is used to set the `sendTimeout` on the `MessagingTemplate` instance. The `replyTimeout` property, on the other hand, is used to set the `receiveTimeout` property on the `MessagingTemplate` instance.

To simulate connection timeouts, you can connect to a non-routable IP address, such as 10.255.255.10.

21.6. HTTP Proxy configuration

If you are behind a proxy and need to configure proxy settings for HTTP outbound adapters or gateways, you can apply one of two approaches. In most cases, you can rely on the standard Java system properties that control the proxy settings. Otherwise, you can explicitly configure a Spring bean for the HTTP client request factory instance.

21.6.1. Standard Java Proxy configuration

You can set three system properties to configure the proxy settings that are used by the HTTP protocol handler:

- `http.proxyHost`: The host name of the proxy server.
- `http.proxyPort`: The port number (the default is `80`).
- `http.nonProxyHosts`: A list of hosts that should be reached directly, bypassing the proxy. This is a list of patterns separated by `|`. The patterns may start or end with a `*` for wildcards. Any host that matches one of these patterns is reached through a direct connection instead of through a proxy.

For HTTPS, the following properties are available:

- `https.proxyHost`: The host name of the proxy server.
- `https.proxyPort`: The port number, the default value being 80.

For more information, see docs.oracle.com/javase/8/docs/technotes/guides/net/proxies.html

21.6.2. Spring's `SimpleClientHttpRequestFactory`

If you need more explicit control over the proxy configuration, you can use Spring's `SimpleClientHttpRequestFactory` and configure its 'proxy' property, as the following example shows:

```
<bean id="requestFactory"
    class="org.springframework.http.client.SimpleClientHttpRequestFactory">
    <property name="proxy">
        <bean id="proxy" class="java.net.Proxy">
            <constructor-arg>
                <util:constant static-field="java.net.Proxy.Type.HTTP"/>
            </constructor-arg>
            <constructor-arg>
                <bean class="java.net.InetSocketAddress">
                    <constructor-arg value="123.0.0.1"/>
                    <constructor-arg value="8080"/>
                </bean>
            </constructor-arg>
        </bean>
    </property>
</bean>
```

21.7. HTTP Header Mappings

Spring Integration provides support for HTTP header mapping for both HTTP Request and HTTP Responses.

By default, all standard HTTP headers are mapped from the message to HTTP request or response headers without further configuration. However, if you do need further customization, you can provide additional configuration by taking advantage of the namespace support. You can provide a comma-separated list of header names, and you can include simple patterns with the '*' character acting as a wildcard. Provide such values overrides the default behavior. Basically, it assumes you are in complete control at that point. However, if you do want to include all of the standard HTTP headers, you can use the shortcut patterns: `HTTP_REQUEST_HEADERS` and `HTTP_RESPONSE_HEADERS`. The following listing shows two examples (the first of which uses a wildcard):

```
<int-http:outbound-gateway id="httpGateway"
    url="http://localhost/test2"
    mapped-request-headers="thing1, thing2"
    mapped-response-headers="X-*, HTTP_RESPONSE_HEADERS"
    channel="someChannel"/>

<int-http:outbound-channel-adapter id="httpAdapter"
    url="http://localhost/test2"
    mapped-request-headers="thing1, thing2, HTTP_REQUEST_HEADERS"
    channel="someChannel"/>
```

The adapters and gateways use the `DefaultHttpHeaderMapper`, which now provides two static factory methods for inbound and outbound adapters so that the proper direction can be applied (mapping HTTP requests and responses either in or out, as appropriate).

If you need further customization, you can also configure a `DefaultHttpHeaderMapper` independently and inject it into the adapter through the `header-mapper` attribute.

Before version 5.0, the `DefaultHttpHeaderMapper` the default prefix for user-defined, non-standard HTTP headers was `X-`. Version 5.0 changed the default prefix to an empty string. According to RFC-6648, the use of such prefixes is now discouraged. You can still customize this option by setting the `DefaultHttpHeaderMapper.setUserDefinedHeaderPrefix()` property. The following example configures a header mapper for an HTTP gateway:

```
<int-http:outbound-gateway id="httpGateway"
    url="http://localhost/test2"
    header-mapper="headerMapper"
    channel="someChannel"/>

<bean id="headerMapper" class="o.s.i.http.support.DefaultHttpHeaderMapper">
    <property name="inboundHeaderNames" value="thing1*, *thing2, thing3"/>
    <property name="outboundHeaderNames" value="a*b, d"/>
</bean>
```

If you need to do something other than what the `DefaultHttpHeaderMapper` supports, you can implement the `HeaderMapper` strategy interface directly and provide a reference to your implementation.

21.8. Integration Graph Controller

Starting with version 4.3, the HTTP module provides an `@EnableIntegrationGraphController` configuration class annotation and an `<int-http:graph-controller/>` XML element to expose the `IntegrationGraphServer` as a REST service. See Integration Graph for more information.

21.9. HTTP Samples

This section wraps up our coverage of Spring Integration's HTTP support with a few examples.

21.9.1. Multipart HTTP Request — RestTemplate (Client) and Http Inbound Gateway (Server)

This example shows how simple it is to send a multipart HTTP request with Spring's `RestTemplate` and receive it with a Spring Integration HTTP inbound adapter. We create a `MultiValueMap` and populate it with multipart data. The `RestTemplate` takes care of the rest (no pun intended) by converting it to a `MultipartHttpServletRequest`. This particular client sends a multipart HTTP Request that contains the name of the company and an image file (the company logo). The following listing shows the example:

```
RestTemplate template = new RestTemplate();
String uri = "http://localhost:8080/multipart-http/inboundAdapter.htm";
Resource s2logo =
   new ClassPathResource("org/springframework/samples/multipart/spring09_logo.png
");
MultiValueMap map = new LinkedMultiValueMap();
map.add("company", "SpringSource");
map.add("company-logo", s2logo);
HttpHeaders headers = new HttpHeaders();
headers.setContentType(new MediaType("multipart", "form-data"));
HttpEntity request = new HttpEntity(map, headers);
ResponseEntity<?> httpResponse = template.exchange(uri, HttpMethod.POST, request,
 null);
```

That is all we need for the client.

On the server side, we have the following configuration:

```
<int-http:inbound-channel-adapter id="httpInboundAdapter"
    channel="receiveChannel"
    path="/inboundAdapter.htm"
    supported-methods="GET, POST"/>

<int:channel id="receiveChannel"/>

<int:service-activator input-channel="receiveChannel">
    <bean class=
"org.springframework.integration.samples.multipart.MultipartReceiver"/>
</int:service-activator>

<bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>
```

The 'httpInboundAdapter' receives the request and converts it to a `Message` with a payload that is a `LinkedMultiValueMap`. We then parse that in the 'multipartReceiver' service-activator, as the following example shows:

```
public void receive(LinkedMultiValueMap<String, Object> multipartRequest){
    System.out.println("### Successfully received multipart request ###");
    for (String elementName : multipartRequest.keySet()) {
        if (elementName.equals("company")){
            System.out.println("\t" + elementName + " - " +
                ((String[]) multipartRequest.getFirst("company"))[0]);
        }
        else if (elementName.equals("company-logo")){
            System.out.println("\t" + elementName + " - as UploadedMultipartFile:
" +
                ((UploadedMultipartFile) multipartRequest
                    .getFirst("company-logo")).getOriginalFilename());
        }
    }
}
```

You should see the following output:

```
### Successfully received multipart request ###
   company - SpringSource
   company-logo - as UploadedMultipartFile: spring09_logo.png
```

Chapter 22. JDBC Support

Spring Integration provides channel adapters for receiving and sending messages by using database queries. Through those adapters, Spring Integration supports not only plain JDBC SQL queries but also stored procedure and stored function calls.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-jdbc</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-jdbc:5.3.0.RC1"
```

By default, the following JDBC components are available:

- Inbound Channel Adapter
- Outbound Channel Adapter
- Outbound Gateway
- Stored Procedure Inbound Channel Adapter
- Stored Procedure Outbound Channel Adapter
- Stored Procedure Outbound Gateway

The Spring Integration JDBC Module also provides a JDBC Message Store.

22.1. Inbound Channel Adapter

The main function of an inbound channel adapter is to execute a SQL `SELECT` query and turn the result set into a message. The message payload is the whole result set (expressed as a `List`), and the types of the items in the list depend on the row-mapping strategy. The default strategy is a generic mapper that returns a `Map` for each row in the query result. Optionally, you can change this by adding a reference to a `RowMapper` instance (see the Spring JDBC documentation for more detailed information about row mapping).

If you want to convert rows in the `SELECT` query result to individual messages, you can use a downstream splitter.

The inbound adapter also requires a reference to either a `JdbcTemplate` instance or a `DataSource`.

As well as the SELECT statement to generate the messages, the adapter also has an UPDATE statement that marks the records as processed so that they do not show up in the next poll. The update can be parameterized by the list of IDs from the original select. By default, this is done through a naming convention (a column in the input result set called id is translated into a list in the parameter map for the update called id). The following example defines an inbound channel adapter with an update query and a DataSource reference.

```
<int-jdbc:inbound-channel-adapter query="select * from item where status=2"
    channel="target" data-source="dataSource"
    update="update item set status=10 where id in (:id)" />
```

The parameters in the update query are specified with a colon (:) prefix to the name of a parameter (which, in the preceding example, is an expression to be applied to each of the rows in the polled result set). This is a standard feature of the named parameter JDBC support in Spring JDBC, combined with a convention (projection onto the polled result list) adopted in Spring Integration. The underlying Spring JDBC features limit the available expressions (for example, most special characters other than a period are disallowed), but since the target is usually a list of objects (possibly a list of one) that are addressable by bean paths this is not unduly restrictive.

To change the parameter generation strategy, you can inject a SqlParameterSourceFactory into the adapter to override the default behavior (the adapter has a sql-parameter-source-factory attribute). Spring Integration provides ExpressionEvaluatingSqlParameterSourceFactory, which creates a SpEL-based parameter source, with the results of the query as the #root object. (If update-per-row is true, the root object is the row). If the same parameter name appears multiple times in the update query, it is evaluated only once, and its result is cached.

You can also use a parameter source for the select query. In this case, since there is no “result” object to evaluate against, a single parameter source is used each time (rather than using a parameter source factory). Starting with version 4.0, you can use Spring to create a SpEL based parameter source, as the following example shows:

```
<int-jdbc:inbound-channel-adapter query="select * from item where status=:status"
    channel="target" data-source="dataSource"
    select-sql-parameter-source="parameterSource" />

<bean id="parameterSource" factory-bean="parameterSourceFactory"
            factory-method="createParameterSourceNoCache">
    <constructor-arg value="" />
</bean>

<bean id="parameterSourceFactory"
        class="o.s.integration.jdbc.ExpressionEvaluatingSqlParameterSourceFactory
">
    <property name="parameterExpressions">
        <map>
            <entry key="status" value="@statusBean.which()" />
        </map>
    </property>
</bean>

<bean id="statusBean" class="foo.StatusDetermination" />
```

The `value` in each parameter expression can be any valid SpEL expression. The `#root` object for the expression evaluation is the constructor argument defined on the `parameterSource` bean. It is static for all evaluations (in the preceding example, an empty `String`).

Starting with version 5.0, you ca supply `ExpressionEvaluatingSqlParameterSourceFactory` with `sqlParameterTypes` to specify the target SQL type for the particular parameter.

The following example provides SQL types for the parameters being used in the query:

```
<int-jdbc:inbound-channel-adapter query="select * from item where status=:status"
    channel="target" data-source="dataSource"
    select-sql-parameter-source="parameterSource" />

<bean id="parameterSource" factory-bean="parameterSourceFactory"
            factory-method="createParameterSourceNoCache">
    <constructor-arg value="" />
</bean>

<bean id="parameterSourceFactory"
        class="o.s.integration.jdbc.ExpressionEvaluatingSqlParameterSourceFactory
">
    <property name="sqlParameterTypes">
        <map>
            <entry key="status" value="#{ T(java.sql.Types).BINARY}" />
        </map>
    </property>
</bean>
```

Use the `createParameterSourceNoCache` factory method. Otherwise, the parameter source caches the result of the evaluation. Also note that, because caching is disabled, if the same parameter name appears in the select query multiple times, it is re-evaluated for each occurrence.

22.1.1. Polling and Transactions

The inbound adapter accepts a regular Spring Integration poller as a child element. Consequently, the frequency of the polling can be controlled (among other uses). An important feature of the poller for JDBC usage is the option to wrap the poll operation in a transaction, as the following example shows:

```
<int-jdbc:inbound-channel-adapter query="..."
        channel="target" data-source="dataSource" update="...">
    <int:poller fixed-rate="1000">
        <int:transactional/>
    </int:poller>
</int-jdbc:inbound-channel-adapter>
```

If you do not explicitly specify a poller, a default value is used. As is normal with Spring Integration, it can be defined as a top-level bean).

In the preceding example, the database is polled every 1000 milliseconds (or once a second), and the update and select queries are both executed in the same transaction. The transaction manager

configuration is not shown. However, as long as it is aware of the data source, the poll is transactional. A common use case is for the downstream channels to be direct channels (the default), so that the endpoints are invoked in the same thread and, hence, the same transaction. That way, if any of them fail, the transaction rolls back and the input data is reverted to its original state.

22.1.2. `max-rows` Versus `max-messages-per-poll`

The JDBC inbound channel adapter defines an attribute called `max-rows`. When you specify the adapter's poller, you can also define a property called `max-messages-per-poll`. While these two attributes look similar, their meaning is quite different.

`max-messages-per-poll` specifies the number of times the query is executed per polling interval, whereas `max-rows` specifies the number of rows returned for each execution.

Under normal circumstances, you would likely not want to set the poller's `max-messages-per-poll` property when you use the JDBC inbound channel adapter. Its default value is `1`, which means that the JDBC inbound channel adapter's `receive()` method is executed exactly once for each poll interval.

Setting the `max-messages-per-poll` attribute to a larger value means that the query is executed that many times back to back. For more information regarding the `max-messages-per-poll` attribute, see Configuring An Inbound Channel Adapter.

In contrast, the `max-rows` attribute, if greater than `0`, specifies the maximum number of rows to be used from the query result set created by the `receive()` method. If the attribute is set to `0`, all rows are included in the resulting message. The attribute defaults to `0`.

It is recommended to use result set limiting via vendor-specific query options, for example MySQL `LIMIT` or SQL Server `TOP` or Oracle's `ROWNUM`. See the particular vendor documentation for more information.

22.2. Outbound Channel Adapter

The outbound channel adapter is the inverse of the inbound: its role is to handle a message and use it to execute a SQL query. By default, the message payload and headers are available as input parameters to the query, as the following example shows:

```
<int-jdbc:outbound-channel-adapter
    query="insert into foos (id, status, name) values (:headers[id], 0,
:payload[something])"
    data-source="dataSource"
    channel="input"/>
```

In the preceding example, messages arriving on the channel labelled `input` have a payload of a map with a key of `something`, so the `[]` operator dereferences that value from the map. The headers are

also accessed as a map.

The parameters in the preceding query are bean property expressions on the incoming message (not SpEL expressions). This behavior is part of the `SqlParameterSource`, which is the default source created by the outbound adapter. You can inject a different `SqlParameterSourceFactory` to get different behavior.

The outbound adapter requires a reference to either a `DataSource` or a `JdbcTemplate`. You can also inject a `SqlParameterSourceFactory` to control the binding of each incoming message to a query.

If the input channel is a direct channel, the outbound adapter runs its query in the same thread and, therefore, the same transaction (if there is one) as the sender of the message.

22.2.1. Passing Parameters by Using SpEL Expressions

A common requirement for most JDBC channel adapters is to pass parameters as part of SQL queries or stored procedures or functions. As mentioned earlier, these parameters are by default bean property expressions, not SpEL expressions. However, if you need to pass SpEL expression as parameters, you must explicitly inject a `SqlParameterSourceFactory`.

The following example uses a `ExpressionEvaluatingSqlParameterSourceFactory` to achieve that requirement:

```
<jdbc:outbound-channel-adapter data-source="dataSource" channel="input"
    query="insert into MESSAGES (MESSAGE_ID,PAYLOAD,CREATED_DATE)     \
    values (:id, :payload, :createdDate)"
    sql-parameter-source-factory="spelSource"/>

<bean id="spelSource"
      class="o.s.integration.jdbc.ExpressionEvaluatingSqlParameterSourceFactory">
    <property name="parameterExpressions">
        <map>
            <entry key="id"          value="headers['id'].toString()"/>
            <entry key="createdDate" value="new java.util.Date()"/>
            <entry key="payload"     value="payload"/>
        </map>
    </property>
</bean>
```

For further information, see Defining Parameter Sources.

22.2.2. Using the `PreparedStatement` Callback

Sometimes, the flexibility and loose-coupling of `SqlParameterSourceFactory` does not do what we need for the target `PreparedStatement` or we need to do some low-level JDBC work. The Spring JDBC module provides APIs to configure the execution environment (such as `ConnectionCallback` or `PreparedStatementCreator`) and manipulate parameter values (such as `SqlParameterSource`). It can

even access APIs for low-level operations, such as `StatementCallback`.

Starting with Spring Integration 4.2, `MessagePreparedStatementSetter` allows the specification of parameters on the `PreparedStatement` manually, in the `requestMessage` context. This class plays exactly the same role as `PreparedStatementSetter` in the standard Spring JDBC API. Actually, it is invoked directly from an inline `PreparedStatementSetter` implementation when the `JdbcMessageHandler` invokes `execute` on the `JdbcTemplate`.

This functional interface option is mutually exclusive with `sqlParameterSourceFactory` and can be used as a more powerful alternative to populate parameters of the `PreparedStatement` from the `requestMessage`. For example, it is useful when we need to store `File` data to the DataBase `BLOB` column in a streaming manner. The following example shows how to do so:

```
@Bean
@ServiceActivator(inputChannel = "storeFileChannel")
public MessageHandler jdbcMessageHandler(DataSource dataSource) {
    JdbcMessageHandler jdbcMessageHandler = new JdbcMessageHandler(dataSource,
            "INSERT INTO imagedb (image_name, content, description) VALUES (?, ?,
?)");
    jdbcMessageHandler.setPreparedStatementSetter((ps, m) -> {
        ps.setString(1, m.getHeaders().get(FileHeaders.FILENAME));
        try (FileInputStream inputStream = new FileInputStream((File) m.
getPayload()); ) {
            ps.setBlob(2, inputStream);
        }
        catch (Exception e) {
            throw new MessageHandlingException(m, e);
        }
        ps.setClob(3, new StringReader(m.getHeaders().get("description", String
.class)));
    });
    return jdbcMessageHandler;
}
```

From the XML configuration perspective, the `prepared-statement-setter` attribute is available on the `<int-jdbc:outbound-channel-adapter>` component. It lets you specify a `MessagePreparedStatementSetter` bean reference.

22.2.3. Batch Update

Starting with version 5.1, the `JdbcMessageHandler` performs a `JdbcOperations.batchUpdate()` if the payload of the request message is an `Iterable` instance. Each element of the `Iterable` is wrapped to a `Message` with the headers from the request message. In the case of regular `SqlParameterSourceFactory`-based configuration these messages are used to build an `SqlParameterSource[]` for an argument used in the mentioned `JdbcOperations.batchUpdate()` function. When a `MessagePreparedStatementSetter` configuration is applied, a `BatchPreparedStatementSetter` variant is used to iterate over those messages for each item and the

provided `MessagePreparedStatementSetter` is called against them. The batch update is not supported when `keysGenerated` mode is selected.

22.3. Outbound Gateway

The outbound gateway is like a combination of the outbound and inbound adapters: Its role is to handle a message and use it to execute a SQL query and then respond with the result by sending it to a reply channel. By default, the message payload and headers are available as input parameters to the query, as the following example shows:

```
<int-jdbc:outbound-gateway
    update="insert into mythings (id, status, name) values (:headers[id], 0,
:payload[thing])"
    request-channel="input" reply-channel="output" data-source="dataSource" />
```

The result of the preceding example is to insert a record into the `mythings` table and return a message that indicates the number of rows affected (the payload is a map: `{UPDATED=1}`) to the output channel .

If the update query is an insert with auto-generated keys, you can populate the reply message with the generated keys by adding `keys-generated="true"` to the preceding example (this is not the default because it is not supported by some database platforms). The following example shows the changed configuration:

```
<int-jdbc:outbound-gateway
    update="insert into mythings (status, name) values (0, :payload[thing])"
    request-channel="input" reply-channel="output" data-source="dataSource"
    keys-generated="true"/>
```

Instead of the update count or the generated keys, you can also provide a select query to execute and generate a reply message from the result (such as the inbound adapter), as the following example shows:

```
<int-jdbc:outbound-gateway
    update="insert into foos (id, status, name) values (:headers[id], 0,
:payload[foo])"
    query="select * from foos where id=:headers[$id]"
    request-channel="input" reply-channel="output" data-source="dataSource"/>
```

Since Spring Integration 2.2, the update SQL query is no longer mandatory. You can now provide only a select query, by using either the `query` attribute or the `query` element. This is extremely useful

if you need to actively retrieve data by using, for example, a generic gateway or a payload enricher. The reply message is then generated from the result (similar to how the inbound adapter works) and passed to the reply channel. The following example show to use the `query` attribute:

```
<int-jdbc:outbound-gateway
    query="select * from foos where id=:headers[id]"
    request-channel="input"
    reply-channel="output"
    data-source="dataSource"/>
```

By default, the component for the `SELECT` query returns only one (the first) row from the cursor. You can adjust this behavior with the `max-rows` option. If you need to return all the rows from the SELECT, consider specifying `max-rows="0"`.

As with the channel adapters, you can also provide `SqlParameterSourceFactory` instances for request and reply. The default is the same as for the outbound adapter, so the request message is available as the root of an expression. If `keys-generated="true"`, the root of the expression is the generated keys (a map if there is only one or a list of maps if multi-valued).

The outbound gateway requires a reference to either a `DataSource` or a `JdbcTemplate`. It can also have a `SqlParameterSourceFactory` injected to control the binding of the incoming message to the query.

Starting with the version 4.2, the `request-prepared-statement-setter` attribute is available on the `<int-jdbc:outbound-gateway>` as an alternative to `request-sql-parameter-source-factory`. It lets you specify a `MessagePreparedStatementSetter` bean reference, which implements more sophisticated `PreparedStatement` preparation before its execution.

See Outbound Channel Adapter for more information about `MessagePreparedStatementSetter`.

22.4. JDBC Message Store

Spring Integration provides two JDBC specific message store implementations. The `JdbcMessageStore` is suitable for use with aggregators and the claim check pattern. The `JdbcChannelMessageStore` implementation provides a more targeted and scalable implementation specifically for message channel.

Note that you can use a `JdbcMessageStore` to back a message channel, `JdbcChannelMessageStore` is optimized for that purpose.

Starting with versions 5.0.11, 5.1.2, the indexes for the `JdbcChannelMessageStore` have been optimized. If you have large message groups in such a store, you may wish to alter the indexes. Furthermore, the index for `PriorityChannel` is commented out because it is not needed unless you are using such channels backed by JDBC.

When using the `OracleChannelMessageStoreQueryProvider`, the priority channel index **must** be added because it is included in a hint in the query.

22.4.1. Initializing the Database

Before starting to use JDBC message store components, you should provision a target database with the appropriate objects.

Spring Integration ships with some sample scripts that can be used to initialize a database. In the `spring-integration-jdbc` JAR file, you can find scripts in the `org.springframework.integration.jdbc` package. It provides an example create and an example drop script for a range of common database platforms. A common way to use these scripts is to reference them in a Spring JDBC data source initializer. Note that the scripts are provided as samples and as specifications of the the required table and column names. You may find that you need to enhance them for production use (for, example, by adding index declarations).

22.4.2. The Generic JDBC Message Store

The JDBC module provides an implementation of the Spring Integration `MessageStore` (important in the claim check pattern) and `MessageGroupStore` (important in stateful patterns such as an aggregator) backed by a database. Both interfaces are implemented by the `JdbcMessageStore`, and there is support for configuring store instances in XML, as the following example shows:

```
<int-jdbc:message-store id="messageStore" data-source="dataSource"/>
```

You can specify a `JdbcTemplate` instead of a `DataSource`.

The following example shows some other optional attributes:

```
<int-jdbc:message-store id="messageStore" data-source="dataSource"
    lob-handler="lobHandler" table-prefix="MY_INT_"/>
```

In the preceding example, we have specified a `LobHandler` for dealing with messages as large objects (which is often necessary for Oracle) and a prefix for the table names in the queries generated by the store. The table name prefix defaults to `INT_`.

22.4.3. Backing Message Channels

If you intend to backing message channels with JDBC, we recommend using the `JdbcChannelMessageStore` implementation. It works only in conjunction with Message Channels.

Supported Databases

The `JdbcChannelMessageStore` uses database-specific SQL queries to retrieve messages from the database. Therefore, you must set the `ChannelMessageStoreQueryProvider` property on the `JdbcChannelMessageStore`. This `channelMessageStoreQueryProvider` provides the SQL queries for the particular database you specify. Spring Integration provides support for the following relational databases:

- PostgreSQL
- HSQLDB
- MySQL
- Oracle
- Derby
- H2
- SqlServer
- Sybase
- DB2

If your database is not listed, you can extend the `AbstractChannelMessageStoreQueryProvider` class and provide your own custom queries.

Version 4.0 added the `MESSAGE_SEQUENCE` column to the table to ensure first-in-first-out (FIFO) queueing even when messages are stored in the same millisecond.

Custom Message Insertion

Since version 5.0, by overloading the `ChannelMessageStorePreparedStatementSetter` class, you can provide a custom implementation for message insertion in the `JdbcChannelMessageStore`. You can use it to set different columns or change the table structure or serialization strategy. For example, instead of default serialization to `byte[]`, you can store its structure as a JSON string.

The following example uses the default implementation of `setValues` to store common columns and overrides the behavior to store the message payload as a `varchar`:

```
public class JsonPreparedStatementSetter extends
ChannelMessageStorePreparedStatementSetter {

    @Override
    public void setValues(PreparedStatement preparedStatement, Message<?>
requestMessage,
        Object groupId, String region,  boolean priorityEnabled) throws
SQLException {
        // Populate common columns
        super.setValues(preparedStatement, requestMessage, groupId, region,
priorityEnabled);
        // Store message payload as varchar
        preparedStatement.setString(6, requestMessage.getPayload().toString());
    }
}
```

Generally, we do not recommend using a relational database for queuing. Instead, if possible, consider using either JMS- or AMQP-backed channels instead. For further reference, see the following resources:

- 5 subtle ways you're using MySQL as a queue, and why it'll bite you.
- The Database As Queue Anti-Pattern.

Concurrent Polling

When polling a message channel, you have the option to configure the associated `Poller` with a `TaskExecutor` reference.

Keep in mind, though, that if you use a JDBC backed message channel and you plan to poll the channel and consequently the message store transactionally with multiple threads, you should ensure that you use a relational database that supports Multiversion Concurrency Control (MVCC). Otherwise, locking may be an issue and the performance, when using multiple threads, may not materialize as expected. For example, Apache Derby is problematic in that regard.

To achieve better JDBC queue throughput and avoid issues when different threads may poll the same `Message` from the queue, it is **important** to set the `usingIdCache` property of `JdbcChannelMessageStore` to `true` when using databases that do not support MVCC. The following example shows how to do so:

```
<bean id="queryProvider"
    class=
"o.s.i.jdbc.store.channel.PostgresChannelMessageStoreQueryProvider"/>

<int:transaction-synchronization-factory id="syncFactory">
    <int:after-commit expression=
"@store.removeFromIdCache(headers.id.toString())" />
    <int:after-rollback expression=
"@store.removeFromIdCache(headers.id.toString())"/>
</int:transaction-synchronization-factory>

<task:executor id="pool" pool-size="10"
    queue-capacity="10" rejection-policy="CALLER_RUNS" />

<bean id="store" class="o.s.i.jdbc.store.JdbcChannelMessageStore">
    <property name="dataSource" ref="dataSource"/>
    <property name="channelMessageStoreQueryProvider" ref=
"queryProvider"/>
    <property name="region" value="TX_TIMEOUT"/>
    <property name="usingIdCache" value="true"/>
</bean>

<int:channel id="inputChannel">
    <int:queue message-store="store"/>
</int:channel>

<int:bridge input-channel="inputChannel" output-channel="outputChannel
">
    <int:poller fixed-delay="500" receive-timeout="500"
        max-messages-per-poll="1" task-executor="pool">
        <int:transactional propagation="REQUIRED" synchronization-
factory="syncFactory"
        isolation="READ_COMMITTED" transaction-manager=
"transactionManager" />
    </int:poller>
</int:bridge>

<int:channel id="outputChannel" />
```

Priority Channel

Starting with version 4.0, `JdbcChannelMessageStore` implements `PriorityCapableChannelMessageStore` and provides the `priorityEnabled` option, letting it be used as a `message-store` reference for `priority-queue` instances. For this purpose, the `INT_CHANNEL_MESSAGE` table has a `MESSAGE_PRIORITY` column to store the value of `PRIORITY` message headers. In addition, a new `MESSAGE_SEQUENCE` column lets us achieve a robust first-in-first-out (FIFO) polling mechanism, even when multiple messages are stored with the same priority in the same millisecond. Messages are polled (selected) from the database with `order by MESSAGE_PRIORITY DESC NULLS LAST, CREATED_DATE, MESSAGE_SEQUENCE`.

We do not recommend using the same `JdbcChannelMessageStore` bean for priority and non-priority queue channels, because the `priorityEnabled` option applies to the entire store and proper FIFO queue semantics are not retained for the queue channel. However, the same `INT_CHANNEL_MESSAGE` table (and even `region`) can be used for both `JdbcChannelMessageStore` types. To configure that scenario, you can extend one message store bean from the other, as the following example shows:

```
<bean id="channelStore" class="o.s.i.jdbc.store.JdbcChannelMessageStore">
    <property name="dataSource" ref="dataSource"/>
    <property name="channelMessageStoreQueryProvider" ref="queryProvider"/>
</bean>

<int:channel id="queueChannel">
    <int:queue message-store="channelStore"/>
</int:channel>

<bean id="priorityStore" parent="channelStore">
    <property name="priorityEnabled" value="true"/>
</bean>

<int:channel id="priorityChannel">
    <int:priority-queue message-store="priorityStore"/>
</int:channel>
```

22.4.4. Partitioning a Message Store

It is common to use a `JdbcMessageStore` as a global store for a group of applications or nodes in the same application. To provide some protection against name clashes and to give control over the database meta-data configuration, the message store lets the tables be partitioned in two ways. One way is to use separate table names, by changing the prefix (as described earlier). The other way is to specify a `region` name for partitioning data within a single table. An important use case for the second approach is when the `MessageStore` is managing persistent queues that back a Spring Integration Message Channel. The message data for a persistent channel is keyed in the store on the channel name. Consequently, if the channel names are not globally unique, the channels can pick up data that is not intended for them. To avoid this danger, you can use the message store `region` to keep data separate for different physical channels that have the same logical name.

22.5. Stored Procedures

In certain situations, plain JDBC support is not sufficient. Maybe you deal with legacy relational database schemas or you have complex data processing needs, but, ultimately, you have to use stored procedures or stored functions. Since Spring Integration 2.1, we provide three components to execute stored procedures or stored functions:

- Stored Procedures Inbound Channel Adapter

- Stored Procedures Outbound Channel Adapter
- Stored Procedures Outbound Gateway

22.5.1. Supported Databases

In order to enable calls to stored procedures and stored functions, the stored procedure components use the `org.springframework.jdbc.core.simple.SimpleJdbcCall` class. Consequently, the following databases are fully supported for executing stored procedures:

- Apache Derby
- DB2
- MySQL
- Microsoft SQL Server
- Oracle
- PostgreSQL
- Sybase

If you want to execute stored functions instead, the following databases are fully supported:

- MySQL
- Microsoft SQL Server
- Oracle
- PostgreSQL

Even though your particular database may not be fully supported, chances are that you can use the stored procedure Spring Integration components quite successfully anyway, provided your RDBMS supports stored procedures or stored functions.

As a matter of fact, some of the provided integration tests use the H2 database. Nevertheless, it is very important to thoroughly test those usage scenarios.

22.5.2. Configuration

The stored procedure components provide full XML Namespace support, and configuring the components is similar as for the general purpose JDBC components discussed earlier.

22.5.3. Common Configuration Attributes

All stored procedure components share certain configuration parameters:

- `auto-startup`: Lifecycle attribute signaling whether this component should be started during application context startup. It defaults to `true`. Optional.
- `data-source`: Reference to a `javax.sql.DataSource`, which is used to access the database. Required.

- `id`: Identifies the underlying Spring bean definition, which is an instance of either `EventDrivenConsumer` or `PollingConsumer`, depending on whether the outbound channel adapter's `channel` attribute references a `SubscribableChannel` or a `PollableChannel`. Optional.
- `ignore-column-meta-data`: For fully supported databases, the underlying `SimpleJdbcCall` class can automatically retrieve the parameter information for the stored procedure or stored function from the JDBC metadata.

  However, if the database does not support metadata lookups or if you need to provide customized parameter definitions, this flag can be set to `true`. It defaults to `false`. Optional.

- `is-function`: If `true`, a SQL Function is called. In that case, the `stored-procedure-name` or `stored-procedure-name-expression` attributes define the name of the called function. It defaults to `false`. Optional.
- `stored-procedure-name`: This attribute specifies the name of the stored procedure. If the `is-function` attribute is set to `true`, this attribute specifies the function name instead. Either this property or `stored-procedure-name-expression` must be specified.
- `stored-procedure-name-expression`: This attribute specifies the name of the stored procedure by using a SpEL expression. By using SpEL, you have access to the full message (if available), including its headers and payload. You can use this attribute to invoke different stored procedures at runtime. For example, you can provide stored procedure names that you would like to execute as a message header. The expression must resolve to a `String`.

  If the `is-function` attribute is set to `true`, this attribute specifies a stored function. Either this property or `stored-procedure-name` must be specified.

- `jdbc-call-operations-cache-size`: Defines the maximum number of cached `SimpleJdbcCallOperations` instances. Basically, for each stored procedure name, a new `SimpleJdbcCallOperations` instance is created that, in return, is cached.

Spring Integration 2.2 added the `stored-procedure-name-expression` attribute and the `jdbc-call-operations-cache-size` attribute.

  The default cache size is `10`. A value of `0` disables caching. Negative values are not permitted.

  If you enable JMX, statistical information about the `jdbc-call-operations-cache` is exposed as an MBean. See MBean Exporter for more information.

- `sql-parameter-source-factory`: (Not available for the stored procedure inbound channel adapter.) Reference to a `SqlParameterSourceFactory`. By default, bean properties of the passed in `Message` payload are used as a source for the stored procedure's input parameters by using a `BeanPropertySqlParameterSourceFactory`.

  This may suffice for basic use cases. For more sophisticated options, consider passing in one or more `ProcedureParameter` values. See Defining Parameter Sources. Optional.

- `use-payload-as-parameter-source`: (Not available for the stored procedure inbound channel adapter.) If set to `true`, the payload of the `Message` is used as a source for providing parameters. If set to `false`, however, the entire `Message` is available as a source for parameters.

If no procedure parameters are passed in, this property defaults to `true`. This means that, by using a default `BeanPropertySqlParameterSourceFactory`, the bean properties of the payload are used as a source for parameter values for the stored procedure or stored function.

However, if procedure parameters are passed in, this property (by default) evaluates to `false`. `ProcedureParameter` lets SpEL Expressions be provided. Therefore, it is highly beneficial to have access to the entire `Message`. The property is set on the underlying `StoredProcExecutor`. Optional.

22.5.4. Common Configuration Sub-Elements

The stored procedure components share a common set of child elements that you can use to define and pass parameters to stored procedures or stored functions. The following elements are available:

- `parameter`
- `returning-resultset`
- `sql-parameter-definition`
- `poller`
- `parameter`: Provides a mechanism to provide stored procedure parameters. Parameters can be either static or provided by using a SpEL Expressions.

```
<int-jdbc:parameter name=""          ①
                    type=""          ②
                    value=""/>       ③

<int-jdbc:parameter name=""
                    expression=""/> ④
```

+ <1> The name of the parameter to be passed into the Stored Procedure or Stored Function. Required. <2> This attribute specifies the type of the value. If nothing is provided, this attribute defaults to `java.lang.String`. This attribute is used only when the `value` attribute is used. Optional. <3> The value of the parameter. You must provide either this attribute or the `expression` attribute. Optional. <4> Instead of the `value` attribute, you can specify a SpEL expression for passing the value of the parameter. If you specify the `expression`, the `value` attribute is not allowed. Optional.

Optional.

- `returning-resultset`: Stored procedures may return multiple result sets. By setting one or more `returning-resultset` elements, you can specify `RowMappers` to convert each returned `ResultSet` to meaningful objects. Optional.

```
<int-jdbc:returning-resultset name="" row-mapper="" />
```

- `sql-parameter-definition`: If you use a database that is fully supported, you typically do not have to specify the stored procedure parameter definitions. Instead, those parameters can be automatically derived from the JDBC metadata. However, if you use databases that are not fully supported, you must set those parameters explicitly by using the `sql-parameter-definition` element.

  You can also choose to turn off any processing of parameter metadata information obtained through JDBC by using the `ignore-column-meta-data` attribute.

```
<int-jdbc:sql-parameter-definition
                                    name=""                          ①
                                    direction="IN"                   ②
                                    type="STRING"                    ③
                                    scale="5"                        ④
                                    type-name="FOO_STRUCT"           ⑤
                                    return-type="fooSqlReturnType"/> ⑥
```

① Specifies the name of the SQL parameter. Required.

② Specifies the direction of the SQL parameter definition. Defaults to `IN`. Valid values are: `IN`, `OUT`, and `INOUT`. If your procedure is returning result sets, use the `returning-resultset` element. Optional.

③ The SQL type used for this SQL parameter definition. Translates into an integer value, as defined by `java.sql.Types`. Alternatively, you can provide the integer value as well. If this attribute is not explicitly set, it defaults to 'VARCHAR'. Optional.

④ The scale of the SQL parameter. Only used for numeric and decimal parameters. Optional.

⑤ The `typeName` for types that are user-named, such as: `STRUCT`, `DISTINCT`, `JAVA_OBJECT`, and named array types. This attribute is mutually exclusive with the `scale` attribute. Optional.

⑥ The reference to a custom value handler for complex types. An implementation of `SqlReturnType`. This attribute is mutually exclusive with the `scale` attribute and is only applicable for OUT and INOUT parameters. Optional.

- `poller`: Lets you configure a message poller if this endpoint is a `PollingConsumer`. Optional.

22.5.5. Defining Parameter Sources

Parameter sources govern the techniques of retrieving and mapping the Spring Integration message properties to the relevant stored procedure input parameters.

The stored procedure components follow certain rules. By default, the bean properties of the `Message` payload are used as a source for the stored procedure's input parameters. In that case, a `BeanPropertySqlParameterSourceFactory` is used. This may suffice for basic use cases. The next example illustrates that default behavior.

For the "automatic" lookup of bean properties by using the `BeanPropertySqlParameterSourceFactory` to work, your bean properties must be defined in lower case. This is due to the fact that in `org.springframework.jdbc.core.metadata.CallMetaDataContext` (the Java method is `matchInParameterValuesWithCallParameters()`), the retrieved stored procedure parameter declarations are converted to lower case. As a result, if you have camel-case bean properties (such as `lastName`), the lookup fails. In that case, provide an explicit `ProcedureParameter`.

Suppose we have a payload that consists of a simple bean with the following three properties: `id`, `name`, and `description`. Furthermore, we have a simplistic Stored Procedure called `INSERT_COFFEE` that accepts three input parameters: `id`, `name`, and `description`. We also use a fully supported database. In that case, the following configuration for a stored procedure outbound adapter suffices:

```
<int-jdbc:stored-proc-outbound-channel-adapter data-source="dataSource"
    channel="insertCoffeeProcedureRequestChannel"
    stored-procedure-name="INSERT_COFFEE"/>
```

For more sophisticated options, consider passing in one or more `ProcedureParameter` values.

If you do provide `ProcedureParameter` values explicitly, by default, an `ExpressionEvaluatingSqlParameterSourceFactory` is used for parameter processing, to enable the full power of SpEL expressions.

If you need even more control over how parameters are retrieved, consider passing in a custom implementation of `SqlParameterSourceFactory` by using the `sql-parameter-source-factory` attribute.

22.5.6. Stored Procedure Inbound Channel Adapter

The following listing calls out the attributes that matter for a stored procedure inbound channel adapter:

```
<int-jdbc:stored-proc-inbound-channel-adapter
                                    channel=""
①
                                    stored-procedure-name=""
                                    data-source=""
                                    auto-startup="true"
                                    id=""
                                    ignore-column-meta-data="false"
                                    is-function="false"
                                    skip-undeclared-results=""
②
                                    return-value-required="false"
③
    <int:poller/>
    <int-jdbc:sql-parameter-definition name="" direction="IN"
                                                type="STRING"
                                                scale=""/>
    <int-jdbc:parameter name="" type="" value=""/>
    <int-jdbc:parameter name="" expression=""/>
    <int-jdbc:returning-resultset name="" row-mapper="" />
</int-jdbc:stored-proc-inbound-channel-adapter>
```

① Channel to which polled messages are sent. If the stored procedure or function does not return any data, the payload of the `Message` is null. Required.

② If this attribute is set to `true`, all results from a stored procedure call that do not have a corresponding `SqlOutParameter` declaration are bypassed. For example, stored procedures can return an update count value, even though your stored procedure declared only a single result parameter. The exact behavior depends on the database implementation. The value is set on the underlying `JdbcTemplate`. The value defaults to `true`. Optional.

③ Indicates whether this procedure's return value should be included. Since Spring Integration 3.0. Optional.

22.5.7. Stored Procedure Outbound Channel Adapter

The following listing calls out the attributes that matter for a stored procedure outbound channel adapter:

```
<int-jdbc:stored-proc-outbound-channel-adapter channel=""
①
                                               stored-procedure-name=""
                                               data-source=""
                                               auto-startup="true"
                                               id=""
                                               ignore-column-meta-data="false"
                                               order=""
②
                                               sql-parameter-source-factory=""
                                               use-payload-as-parameter-source="">
    <int:poller fixed-rate=""/>
    <int-jdbc:sql-parameter-definition name=""/>
    <int-jdbc:parameter name=""/>

</int-jdbc:stored-proc-outbound-channel-adapter>
```

① The receiving message channel of this endpoint. Required.

② Specifies the order for invocation when this endpoint is connected as a subscriber to a channel. This is particularly relevant when that channel is using a `failover` dispatching strategy. It has no effect when this endpoint is itself a polling consumer for a channel with a queue. Optional.

22.5.8. Stored Procedure Outbound Gateway

The following listing calls out the attributes that matter for a stored procedure outbound channel adapter:

```
<int-jdbc:stored-proc-outbound-gateway request-channel=""
①
                                      stored-procedure-name=""
                                      data-source=""
                                    auto-startup="true"
                                    id=""
                                    ignore-column-meta-data="false"
                                    is-function="false"
                                    order=""
                                    reply-channel=""
②
                                    reply-timeout=""
③
                                    return-value-required="false"
④
                                    skip-undeclared-results=""
⑤
                                    sql-parameter-source-factory=""
                                    use-payload-as-parameter-source="">
<int-jdbc:sql-parameter-definition name="" direction="IN"
                                    type=""
                                    scale="10"/>
<int-jdbc:sql-parameter-definition name=""/>
<int-jdbc:parameter name="" type="" value=""/>
<int-jdbc:parameter name="" expression=""/>
<int-jdbc:returning-resultset name="" row-mapper="" />
```

① The receiving message channel of this endpoint. Required.

② Message channel to which replies should be sent after receiving the database response. Optional.

③ Lets you specify how long this gateway waits for the reply message to be sent successfully before throwing an exception. Keep in mind that, when sending to a `DirectChannel`, the invocation occurs in the sender's thread. Consequently, the failing of the send operation may be caused by other components further downstream. By default, the gateway waits indefinitely. The value is specified in milliseconds. Optional.

④ Indicates whether this procedure's return value should be included. Optional.

⑤ If the `skip-undeclared-results` attribute is set to `true`, all results from a stored procedure call that do not have a corresponding `SqlOutParameter` declaration are bypassed. For example, stored procedures may return an update count value, even though your stored procedure only declared a single result parameter. The exact behavior depends on the database. The value is set on the underlying `JdbcTemplate`. The value defaults to `true`. Optional.

22.5.9. Examples

This section contains two examples that call Apache Derby stored procedures. The first procedure calls a stored procedure that returns a `ResultSet`. By using a `RowMapper`, the data is converted into a domain object, which then becomes the Spring Integration message payload.

In the second sample, we call a stored procedure that uses output parameters to return data instead.

Have a look at the Spring Integration Samples project.

The project contains the Apache Derby example referenced here, as well as instructions on how to run it. The Spring Integration Samples project also provides an example of using Oracle stored procedures.

In the first example, we call a stored procedure named `FIND_ALL_COFFEE_BEVERAGES` that does not define any input parameters but that returns a `ResultSet`.

In Apache Derby, stored procedures are implemented in Java. The following listing shows the method signature:

```
public static void findAllCoffeeBeverages(ResultSet[] coffeeBeverages)
            throws SQLException {
    ...
}
```

The following listing shows the corresponding SQL:

```
CREATE PROCEDURE FIND_ALL_COFFEE_BEVERAGES() \
PARAMETER STYLE JAVA LANGUAGE JAVA MODIFIES SQL DATA DYNAMIC RESULT SETS 1 \
EXTERNAL NAME
'o.s.i.jdbc.storedproc.derby.DerbyStoredProcedures.findAllCoffeeBeverages';
```

In Spring Integration, you can now call this stored procedure by using, for example, a `stored-proc-outbound-gateway`, as the following example shows:

```
<int-jdbc:stored-proc-outbound-gateway id="outbound-gateway-storedproc-find-all"
                                       data-source="dataSource"
                                       request-channel=
"findAllProcedureRequestChannel"
                                       expect-single-result="true"
                                       stored-procedure-name=
"FIND_ALL_COFFEE_BEVERAGES">
<int-jdbc:returning-resultset name="coffeeBeverages"
    row-mapper="org.springframework.integration.support.CoffeBeverageMapper"/>
</int-jdbc:stored-proc-outbound-gateway>
```

In the second example, we call a stored procedure named `FIND_COFFEE` that has one input parameter. Instead of returning a `ResultSet`, it uses an output parameter. The following example shows the method signature:

```
public static void findCoffee(int coffeeId, String[] coffeeDescription)
            throws SQLException {
    ...
}
```

The following listing shows the corresponding SQL:

```
CREATE PROCEDURE FIND_COFFEE(IN ID INTEGER, OUT COFFEE_DESCRIPTION VARCHAR(200)) \
PARAMETER STYLE JAVA LANGUAGE JAVA EXTERNAL NAME \
'org.springframework.integration.jdbc.storedproc.derby.DerbyStoredProcedures.findC
offee';
```

In Spring Integration, you can now call this Stored Procedure by using, for example, a `stored-proc-outbound-gateway`, as the following example shows:

```
<int-jdbc:stored-proc-outbound-gateway id="outbound-gateway-storedproc-find-
coffee"
                                      data-source="dataSource"
                                      request-channel=
"findCoffeeProcedureRequestChannel"
                                      skip-undeclared-results="true"
                                      stored-procedure-name="FIND_COFFEE"
                                      expect-single-result="true">
    <int-jdbc:parameter name="ID" expression="payload" />
</int-jdbc:stored-proc-outbound-gateway>
```

22.6. JDBC Lock Registry

Version 4.3 introduced the `JdbcLockRegistry`. Certain components (for example, aggregator and resequencer) use a lock obtained from a `LockRegistry` instance to ensure that only one thread manipulates a group at a time. The `DefaultLockRegistry` performs this function within a single component. You can now configure an external lock registry on these components. When used with a shared `MessageGroupStore`, you can use the `JdbcLockRegistry` to provide this functionality across multiple application instances, such that only one instance can manipulate the group at a time.

When a lock is released by a local thread, another local thread can generally acquire the lock immediately. If a lock is released by a thread that uses a different registry instance, it can take up to 100ms to acquire the lock.

The `JdbcLockRegistry` is based on the `LockRepository` abstraction, which has a `DefaultLockRepository` implementation. The database schema scripts are located in the `org.springframework.integration.jdbc` package, which is divided for the particular RDBMS vendors. For example, the following listing shows the H2 DDL for the lock table:

```
CREATE TABLE INT_LOCK  (
    LOCK_KEY CHAR(36),
    REGION VARCHAR(100),
    CLIENT_ID CHAR(36),
    CREATED_DATE TIMESTAMP NOT NULL,
    constraint INT_LOCK_PK primary key (LOCK_KEY, REGION)
);
```

The `INT_` can be changed according to the target database design requirements. Therefore, you must use `prefix` property on the `DefaultLockRepository` bean definition.

Sometimes, one application has moved to such a state that it cannot release the distributed lock and remove the particular record in the database. For this purpose, such dead locks can be expired by the other application on the next locking invocation. The `timeToLive` (TTL) option on the `DefaultLockRepository` is provided for this purpose. You may also want to specify `CLIENT_ID` for the

locks stored for a given `DefaultLockRepository` instance. If so, you can specify the `id` to be associated with the `DefaultLockRepository` as a constructor parameter.

Starting with version 5.1.8, the `JdbcLockRegistry` can be configured with the `idleBetweenTries` - a `Duration` to sleep between lock record insert/update executions. By default it is `100` milliseconds and in some environments non-leaders pollute connections with data source too often.

22.7. JDBC Metadata Store

Version 5.0 introduced the JDBC `MetadataStore` (see Metadata Store) implementation. You can use the `JdbcMetadataStore` to maintain the metadata state across application restarts. This `MetadataStore` implementation can be used with adapters such as the following:

- Feed inbound channel adapters
- File inbound channel adapters
- FTP inbound channel adapters
- SFTP inbound channel adapters

To configure these adapters to use the `JdbcMetadataStore`, declare a Spring bean by using a bean name of `metadataStore`. The Feed inbound channel adapter and the feed inbound channel adapter both automatically pick up and use the declared `JdbcMetadataStore`, as the following example shows:

```
@Bean
public MetadataStore metadataStore(DataSource dataSource) {
    return new JdbcMetadataStore(dataSource);
}
```

The `org.springframework.integration.jdbc` package has Database schema scripts for several RDMBS vendors. For example, the following listing shows the H2 DDL for the metadata table:

```
CREATE TABLE INT_METADATA_STORE  (
    METADATA_KEY VARCHAR(255) NOT NULL,
    METADATA_VALUE VARCHAR(4000),
    REGION VARCHAR(100) NOT NULL,
    constraint INT_METADATA_STORE_PK primary key (METADATA_KEY, REGION)
);
```

You can change the `INT_` prefix to match the target database design requirements. You can also configure `JdbcMetadataStore` to use the custom prefix.

The `JdbcMetadataStore` implements `ConcurrentMetadataStore`, letting it be reliably shared across multiple application instances, where only one instance can store or modify a key's value. All of

these operations are atomic, thanks to transaction guarantees.

Transaction management must use `JdbcMetadataStore`. Inbound channel adapters can be supplied with a reference to the `TransactionManager` in the poller configuration. Unlike non-transactional `MetadataStore` implementations, with `JdbcMetadataStore`, the entry appears in the target table only after the transaction commits. When a rollback occurs, no entries are added to the `INT_METADATA_STORE` table.

Since version 5.0.7, you can configure the `JdbcMetadataStore` with the RDBMS vendor-specific `lockHint` option for lock-based queries on metadata store entries. By default, it is `FOR UPDATE` and can be configured with an empty string if the target database does not support row locking functionality. Consult with your vendor for particular and possible hints in the `SELECT` expression for locking rows before updates.

Chapter 23. JPA Support

Spring Integration's JPA (Java Persistence API) module provides components for performing various database operations using JPA.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-jpa</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-jpa:5.3.0.RC1"
```

The JPA API must be included via some vendor-specific implementation, e.g. Hibernate ORM Framework.

The following components are provided:

- Inbound channel adapter
- Outbound channel adapter
- Updating outbound gateway
- Retrieving outbound gateway

These components can be used to perform `select`, `create`, `update`, and `delete` operations on the target databases by sending and receiving messages to them.

The JPA inbound channel adapter lets you poll and retrieve (`select`) data from the database by using JPA, whereas the JPA outbound channel adapter lets you create, update, and delete entities.

You can use outbound gateways for JPA to persist entities to the database, letting you continue the flow and execute further components downstream. Similarly, you can use an outbound gateway to retrieve entities from the database.

For example, you may use the outbound gateway, which receives a `Message` with a `userId` as payload on its request channel, to query the database, retrieve the user entity, and pass it downstream for further processing.

Recognizing these semantic differences, Spring Integration provides two separate JPA outbound gateways:

- Retrieving outbound gateway
- Updating outbound gateway

23.1. Functionality

All JPA components perform their respective JPA operations by using one of the following:

- Entity classes
- Java Persistence Query Language (JPQL) for update, select and delete (JPQL does not support inserts)
- Native Query
- Named Query

The following sections describe each of these components in more detail.

23.2. Supported Persistence Providers

The Spring Integration JPA support has been tested against the following persistence providers:

- Hibernate
- EclipseLink

When using a persistence provider, you should ensure that the provider is compatible with JPA 2.1.

23.3. Java Implementation

Each of the provided components uses the `o.s.i.jpa.core.JpaExecutor` class, which, in turn, uses an implementation of the `o.s.i.jpa.core.JpaOperations` interface. `JpaOperations` operates like a typical Data Access Object (DAO) and provides methods such as find, persist, executeUpdate, and so on. For most use cases, the default implementation (`o.s.i.jpa.core.DefaultJpaOperations`) should be sufficient. However, you can specify your own implementation if you require custom behavior.

To initialize a `JpaExecutor`, you must use one of the constructors that accept one of:

- EntityManagerFactory
- EntityManager
- JpaOperations

The following example shows how to initialize a `JpaExecutor` with an `entityManagerFactory` and use it in an outbound gateway:

```
@Bean
public JpaExecutor jpaExecutor() {
    JpaExecutor executor = new JpaExecutor(this.entityManagerFactory);
    executor.setJpaParameters(Collections.singletonList(new JpaParameter("firstName",
null, "#this")));
    executor.setUsePayloadAsParameterSource(true);
    executor.setExpectSingleResult(true);
    return executor;
}

@ServiceActivator(inputChannel = "getEntityChannel")
@Bean
public MessageHandler retrievingJpaGateway() {
    JpaOutboundGateway gateway = new JpaOutboundGateway(jpaExecutor());
    gateway.setGatewayType(OutboundGatewayType.RETRIEVING);
    gateway.setOutputChannelName("resultsChannel");
    return gateway;
}
```

23.4. Namespace Support

When using XML namespace support, the underlying parser classes instantiate the relevant Java classes for you. Thus, you typically need not deal with the inner workings of the JPA adapter. This section documents the XML namespace support provided by Spring Integration and shows you how to use the XML Namespace support to configure the JPA components.

23.4.1. Common XML Namespace Configuration Attributes

Certain configuration parameters are shared by all JPA components:

`auto-startup`

Lifecycle attribute that signals whether this component should be started during application context startup. Defaults to `true`. Optional.

`id`

Identifies the underlying Spring bean definition, which is an instance of either `EventDrivenConsumer` or `PollingConsumer`. Optional.

`entity-manager-factory`

The reference to the JPA entity manager factory that the adapter uses to create the `EntityManager`. You must provide this attribute, the `entity-manager` attribute, or the `jpa-operations` attribute.

`entity-manager`

The reference to the JPA Entity Manager that the component uses. You must provide this attribute, the `entity-manager-factory` attribute, or the `jpa-operations` attribute.

Usually, your Spring application context defines only a JPA entity manager factory, and the `EntityManager` is injected by using the `@PersistenceContext` annotation. This approach does not apply for the Spring Integration JPA components. Usually, injecting the JPA entity manager factory is best, but, when you want to inject an `EntityManager` explicitly, you have to define a `SharedEntityManagerBean`. For more information, see the relevant Javadoc.

The following example shows how to explicitly include an entity manager factory:

```
<bean id="entityManager"
      class="org.springframework.orm.jpa.support.SharedEntityManagerBean">
    <property name="entityManagerFactory" ref="entityManagerFactoryBean" />
</bean>
```

`jpa-operations`

A reference to a bean that implements the `JpaOperations` interface. In rare cases, it might be advisable to provide your own implementation of the `JpaOperations` interface instead of relying on the default implementation (`org.springframework.integration.jpa.core.DefaultJpaOperations`). If you use the `jpa-operations` attribute, you must not provide the JPA entity manager or JPA entity manager factory, because `JpaOperations` wraps the necessary datasource.

`entity-class`

The fully qualified name of the entity class. The exact semantics of this attribute vary, depending on whether we are performing a persist or update operation or whether we are retrieving objects from the database.

When retrieving data, you can specify the `entity-class` attribute to indicate that you would like to retrieve objects of this type from the database. In that case, you must not define any of the query attributes (`jpa-query`, `native-query`, or `named-query`).

When persisting data, the `entity-class` attribute indicates the type of object to persist. If not specified (for persist operations) the entity class is automatically retrieved from the message's payload.

`jpa-query`

Defines the JPA query (Java Persistence Query Language) to be used.

`native-query`

Defines the native SQL query to be used.

`named-query`

Refers to a named query. A named query can be defined in either Native SQL or JPAQL, but the underlying JPA persistence provider handles that distinction internally.

23.4.2. Providing JPA Query Parameters

To provide parameters, you can use the `parameter` XML element. It has a mechanism that lets you provide parameters for queries that are based on either the Java Persistence Query Language (JPQL) or native SQL queries. You can also provide parameters for named queries.

Expression-based Parameters

The following example shows how to set an expression-based parameter:

```
<int-jpa:parameter expression="payload.name" name="firstName"/>
```

Value-based Parameters

The following example shows how to set an value-based parameter:

```
<int-jpa:parameter name="name" type="java.lang.String" value="myName"/>
```

Positional Parameters

The following example shows how to set an expression-based parameter:

```
<int-jpa:parameter expression="payload.name"/>
<int-jpa:parameter type="java.lang.Integer" value="21"/>
```

23.4.3. Transaction Handling

All JPA operations (such as `INSERT`, `UPDATE`, and `DELETE`) require a transaction to be active whenever they are performed. For inbound channel adapters, you need do nothing special. It works similarly to the way we configure transaction managers with pollers that are used with other inbound channel adapters. The following XML example configures a transaction manager that uses a poller with an inbound channel adapter:

```
<int-jpa:inbound-channel-adapter
    channel="inboundChannelAdapterOne"
    entity-manager="em"
    auto-startup="true"
    jpa-query="select s from Student s"
    expect-single-result="true"
    delete-after-poll="true">
    <int:poller fixed-rate="2000" >
        <int:transactional propagation="REQUIRED"
            transaction-manager="transactionManager"/>
    </int:poller>
</int-jpa:inbound-channel-adapter>
```

However, you may need to specifically start a transaction when using an outbound channel adapter or gateway. If a `DirectChannel` is an input channel for the outbound adapter or gateway and if the transaction is active in the current thread of execution, the JPA operation is performed in the same transaction context. You can also configure this JPA operation to run as a new transaction, as the following example shows:

```
<int-jpa:outbound-gateway
    request-channel="namedQueryRequestChannel"
    reply-channel="namedQueryResponseChannel"
    named-query="updateStudentByRollNumber"
    entity-manager="em"
    gateway-type="UPDATING">
    <int-jpa:parameter name="lastName" expression="payload"/>
    <int-jpa:parameter name="rollNumber" expression="headers['rollNumber']"/>
        <int-jpa:transactional propagation="REQUIRES_NEW"
        transaction-manager="transactionManager"/>
</int-jpa:outbound-gateway>
```

In the preceding example, the transactional element of the outbound gateway or adapter specifies the transaction attributes. It is optional to define this child element if you have `DirectChannel` as an input channel to the adapter and you want the adapter to execute the operations in the same transaction context as the caller. If, however, you use an `ExecutorChannel`, you must have the `transactional` element, because the invoking client's transaction context is not propagated.

Unlike the `transactional` element of the poller, which is defined in Spring Integration's namespace, the `transactional` element for the outbound gateway or adapter is defined in the JPA namespace.

23.5. Inbound Channel Adapter

An inbound channel adapter is used to execute a select query over the database using JPA QL and return the result. The message payload is either a single entity or a `List` of entities. The following XML configures an `inbound-channel-adapter`:

```
<int-jpa:inbound-channel-adapter channel="inboundChannelAdapterOne"  ①
                    entity-manager="em"                              ②
                    auto-startup="true"                              ③
                    query="select s from Student s"                  ④
                    expect-single-result="true"                      ⑤
                    max-results=""                                   ⑥
                    max-results-expression=""                        ⑦
                    delete-after-poll="true"                         ⑧
                    flush-after-delete="true">                       ⑨
    <int:poller fixed-rate="2000" >
      <int:transactional propagation="REQUIRED" transaction-manager=
"transactionManager"/>
    </int:poller>
</int-jpa:inbound-channel-adapter>
```

① The channel over which the `inbound-channel-adapter` puts the messages (with the payload) after executing the JPA QL in the `query` attribute.

② The `EntityManager` instance used to perform the required JPA operations.

③ Attribute signaling whether the component should automatically start when the application context starts. The value defaults to `true`.

④ The JPA QL whose result are sent out as the payload of the message

⑤ This attribute tells whether the JPQL query gives a single entity in the result or a `List` of entities. If the value is set to `true`, the single entity is sent as the payload of the message. If, however, multiple results are returned after setting this to `true`, a `MessagingException` is thrown. The value defaults to `false`.

⑥ This non-zero, non-negative integer value tells the adapter not to select more than the given number of rows on execution of the select operation. By default, if this attribute is not set, all possible records are selected by the query. This attribute is mutually exclusive with `max-results-expression`. Optional.

⑦ An expression that is evaluated to find the maximum number of results in a result set. Mutually exclusive with `max-results`. Optional.

⑧ Set this value to `true` if you want to delete the rows received after execution of the query. You must ensure that the component operates as part of a transaction. Otherwise, you may encounter an exception such as: `java.lang.IllegalArgumentException: Removing a detached instance …`

⑨ Set this value to `true` if you want to flush the persistence context immediately after deleting received entities and if you do not want to rely on the `flushMode` of the `EntityManager`. The value defaults to `false`.

23.5.1. Configuration Parameter Reference

The following listing shows all the values that can be set for an `inbound-channel-adapter`:

```
<int-jpa:inbound-channel-adapter
  auto-startup="true"           ①
  channel=""                    ②
  delete-after-poll="false"     ③
  delete-per-row="false"        ④
  entity-class=""               ⑤
  entity-manager=""             ⑥
  entity-manager-factory=""     ⑦
  expect-single-result="false"  ⑧
  id=""
  jpa-operations=""             ⑨
  jpa-query=""                  ⑩
  named-query=""                ⑪
  native-query=""               ⑫
  parameter-source=""           ⑬
  send-timeout="">              ⑭
  <int:poller ref="myPoller"/>
 </int-jpa:inbound-channel-adapter>
```

① This lifecycle attribute signals whether this component should automatically start when the application context starts. This attribute defaults to `true`. Optional.

② The channel to which the adapter sends a message with the payload from performing the desired JPA operation.

③ A boolean flag that indicates whether to delete the selected records after they have been polled by the adapter. By default, the value is `false` (that is, the records are not deleted). You must ensure that the component operates as part of a transaction. Otherwise, you may encounter an exception, such as: `java.lang.IllegalArgumentException: Removing a detached instance …`. Optional.

④ A boolean flag that indicates whether the records can be deleted in bulk or must be deleted one record at a time. By default the value is `false` (that is, the records can be bulk-deleted). Optional.

⑤ The fully qualified name of the entity class to be queried from the database. The adapter automatically builds a JPA Query based on the entity class name. Optional.

⑥ An instance of `javax.persistence.EntityManager` used to perform the JPA operations. Optional.

⑦ An instance of `javax.persistence.EntityManagerFactory` used to obtain an instance of `javax.persistence.EntityManager` that performs the JPA operations. Optional.

⑧ A boolean flag indicating whether the select operation is expected to return a single result or a `List` of results. If this flag is set to `true`, the single entity selected is sent as the payload of the message. If multiple entities are returned, an exception is thrown. If `false`, the `List` of entities is sent as the payload of the message. The value defaults to `false`. Optional.

⑨ An implementation of `org.springframework.integration.jpa.core.JpaOperations` used to perform the JPA operations. We recommend not providing an implementation of your own but using the default `org.springframework.integration.jpa.core.DefaultJpaOperations`

implementation. You can use any of the `entity-manager`, `entity-manager-factory`, or `jpa-operations` attributes. Optional.

⑩ The JPA QL to be executed by this adapter. Optional.

⑪ The named query that needs to be executed by this adapter. Optional.

⑫ The native query executed by this adapter. You can use any of the `jpa-query`, `named-query`, `entity-class`, or `native-query` attributes. Optional.

⑬ An implementation of `o.s.i.jpa.support.parametersource.ParameterSource` used to resolve the values of the parameters in the query. Ignored if the `entity-class` attribute has a value. Optional.

⑭ Maximum amount of time (in milliseconds) to wait when sending a message to the channel. Optional.

23.5.2. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the inbound adapter with Java:

```
@SpringBootApplication
@EntityScan(basePackageClasses = StudentDomain.class)
public class JpaJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(JpaJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Autowired
    private EntityManagerFactory entityManagerFactory;

    @Bean
    public JpaExecutor jpaExecutor() {
        JpaExecutor executor = new JpaExecutor(this.entityManagerFactory);
        jpaExecutor.setJpaQuery("from Student");
        return executor;
    }

    @Bean
    @InboundChannelAdapter(channel = "jpaInputChannel",
                    poller = @Poller(fixedDelay = "${poller.interval}"))
    public MessageSource<?> jpaInbound() {
        return new JpaPollingChannelAdapter(jpaExecutor());
    }

    @Bean
    @ServiceActivator(inputChannel = "jpaInputChannel")
    public MessageHandler handler() {
        return message -> System.out.println(message.getPayload());
    }

}
```

23.5.3. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the inbound adapter with the Java DSL:

```
@SpringBootApplication
@EntityScan(basePackageClasses = StudentDomain.class)
public class JpaJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(JpaJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Autowired
    private EntityManagerFactory entityManagerFactory;

    @Bean
    public IntegrationFlow pollingAdapterFlow() {
        return IntegrationFlows
            .from(Jpa.inboundAdapter(this.entityManagerFactory)
                        .entityClass(StudentDomain.class)
                        .maxResults(1)
                        .expectSingleResult(true),
                e -> e.poller(p -> p.trigger(new OnlyOnceTrigger())))
            .channel(c -> c.queue("pollingResults"))
            .get();
    }

}
```

23.6. Outbound Channel Adapter

The JPA outbound channel adapter lets you accept messages over a request channel. The payload can either be used as the entity to be persisted or used with the headers in the parameter expressions for a JPQL query. The following sections cover the possible ways of performing these operations.

23.6.1. Using an Entity Class

The following XML configures the outbound channel adapter to persist an entity to the database:

```
<int-jpa:outbound-channel-adapter channel="entityTypeChannel"                ①
    entity-class="org.springframework.integration.jpa.test.entity.Student"  ②
    persist-mode="PERSIST"                                                  ③
    entity-manager="em"/ >                                                  ④
```

① The channel over which a valid JPA entity is sent to the JPA outbound channel adapter.

② The fully qualified name of the entity class accepted by the adapter to be persisted in the database. You can actually leave off this attribute in most cases as the adapter can determine the entity class automatically from the Spring Integration message payload.

③ The operation to be done by the adapter. The valid values are `PERSIST`, `MERGE`, and `DELETE`. The default value is `MERGE`.

④ The JPA entity manager to be used.

These four attributes of the `outbound-channel-adapter` configure it to accept entities over the input channel and process them to `PERSIST`, `MERGE`, or `DELETE` the entities from the underlying data source.

As of Spring Integration 3.0, payloads to `PERSIST` or `MERGE` can also be of type `java.lang.Iterable`. In that case, each object returned by the `Iterable` is treated as an entity and persisted or merged using the underlying `EntityManager`. Null values returned by the iterator are ignored.

23.6.2. Using JPA Query Language (JPA QL)

The previous section showed how to perform a `PERSIST` action by using an entity. This section shows how to use an outbound channel adapter with JPA QL.

The following XML configures the outbound channel adapter to persist an entity to the database:

```
<int-jpa:outbound-channel-adapter channel="jpaQlChannel"
①
  jpa-query="update Student s set s.firstName = :firstName where s.rollNumber =
:rollNumber"  ②
  entity-manager="em">
③
    <int-jpa:parameter name="firstName"  expression="payload['firstName']"/>
④
    <int-jpa:parameter name="rollNumber" expression="payload['rollNumber']"/>
</int-jpa:outbound-channel-adapter>
```

① The input channel over which the message is sent to the outbound channel adapter.

② The JPA QL to execute. This query may contain parameters that are evaluated by using the `parameter` element.

③ The entity manager used by the adapter to perform the JPA operations.

④ The elements (one for each parameter) used to define the value of the parameter names for the JPA QL specified in the `query` attribute.

The `parameter` element accepts an attribute whose `name` corresponds to the named parameter specified in the provided JPA QL (point 2 in the preceding example). The value of the parameter can either be static or be derived by using an expression. The static value and the expression to derive the value are specified using the `value` and `expression` attributes, respectively. These attributes are mutually exclusive.

If the `value` attribute is specified, you can provide an optional `type` attribute. The value of this attribute is the fully qualified name of the class whose value is represented by the `value` attribute. By default, the type is assumed to be a `java.lang.String`. The following example shows how to define a JPA parameter:

```
<int-jpa:outbound-channel-adapter ...
>
    <int-jpa:parameter name="level" value="2" type="java.lang.Integer"/>
    <int-jpa:parameter name="name" expression="payload['name']"/>
</int-jpa:outbound-channel-adapter>
```

As the preceding example shows, you can use multiple `parameter` elements within an outbound channel adapter element and define some parameters by using expressions and others with static values. However, take care not to specify the same parameter name multiple times. You should provide one `parameter` element for each named parameter specified in the JPA query. For example, we specify two parameters: `level` and `name`. The `level` attribute is a static value of type `java.lang.Integer`, while the `name` attribute is derived from the payload of the message.

Though specifying `select` is valid for JPA QL, it makes no sense to do so. Outbound channel adapters do not return any result. If you want to select some values, consider using the outbound gateway instead.

23.6.3. Using Native Queries

This section describes how to use native queries to perform operations with the JPA outbound channel adapter. Using native queries is similar to using JPA QL, except that the queries are native database queries. By using native queries, we lose database vendor independence, which we get using JPA QL.

One of the things we can achieve by using native queries is to perform database inserts, which is not possible with JPA QL. (To perform inserts, we send JPA entities to the channel adapter, as described earlier). Below is a small xml fragment that demonstrates the use of native query to insert values in a table.

Named parameters may not be supported by your JPA provider in conjunction with native SQL queries. While they work fine with Hibernate, OpenJPA and EclipseLink do not support them. See issues.apache.org/jira/browse/OPENJPA-111. Section 3.8.12 of the JPA 2.0 spec states: “Only positional parameter binding and positional access to result items may be portably used for native queries.”

The following example configures an outbound-channel-adapter with a native query:

```
<int-jpa:outbound-channel-adapter channel="nativeQlChannel"
  native-query="insert into STUDENT_TABLE(FIRST_NAME,LAST_UPDATED) values
(:lastName,:lastUpdated)"  ①
  entity-manager="em">
    <int-jpa:parameter name="lastName" expression="payload['updatedLastName']"/>
    <int-jpa:parameter name="lastUpdated" expression="new java.util.Date()"/>
</int-jpa:outbound-channel-adapter>
```

① The native query executed by this outbound channel adapter.

Note that the other attributes (such as `channel` and `entity-manager`) and the `parameter` element have the same semantics as they do for JPA QL.

23.6.4. Using Named Queries

Using named queries is similar to using JPA QL or a native query, except that we specify a named query instead of a query. First, we cover how to define a JPA named query. Then we cover how to declare an outbound channel adapter to work with a named query. If we have an entity called `Student`, we can use annotations on the `Student` class to define two named queries: `selectStudent` and `updateStudent`. The following example shows how to do so:

```
@Entity
@Table(name="Student")
@NamedQueries({
    @NamedQuery(name="selectStudent",
        query="select s from Student s where s.lastName = 'Last One'"),
    @NamedQuery(name="updateStudent",
        query="update Student s set s.lastName = :lastName,
               lastUpdated = :lastUpdated where s.id in (select max(a.id) from
Student a)")
})
public class Student {

...
}
```

Alternatively, you can use orm.xml to define named queries as the following example shows:

```
<entity-mappings ...>
    ...
    <named-query name="selectStudent">
        <query>select s from Student s where s.lastName = 'Last One'</query>
    </named-query>
</entity-mappings>
```

Now that we have shown how to define named queries by using annotations or by using `orm.xml`, we now show a small XML fragment that defines an `outbound-channel-adapter` by using a named query, as the following example shows:

```
<int-jpa:outbound-channel-adapter channel="namedQueryChannel"
            named-query="updateStudent"  ①
            entity-manager="em">
        <int-jpa:parameter name="lastName" expression="payload['updatedLastName']
"/>
        <int-jpa:parameter name="lastUpdated" expression="new java.util.Date()"/>
</int-jpa:outbound-channel-adapter>
```

① The named query that we want the adapter to execute when it receives a message over the channel.

23.6.5. Configuration Parameter Reference

The following listing shows all the attributes that you can set on an outbound channel adapter:

```
<int-jpa:outbound-channel-adapter
  auto-startup="true"  ①
  channel=""  ②
  entity-class=""  ③
  entity-manager=""  ④
  entity-manager-factory=""  ⑤
  id=""
  jpa-operations=""  ⑥
  jpa-query=""  ⑦
  named-query=""  ⑧
  native-query=""  ⑨
  order=""  ⑩
  parameter-source-factory=""  ⑪
  persist-mode="MERGE"  ⑫
  flush="true"  ⑬
  flush-size="10"  ⑭
  clear-on-flush="true"  ⑮
  use-payload-as-parameter-source="true"  ⑯
    <int:poller/>
    <int-jpa:transactional/>  ⑰
    <int-jpa:parameter/>  ⑱
</int-jpa:outbound-channel-adapter>
```

① Lifecycle attribute signaling whether this component should start during application context startup. It defaults to `true`. Optional.

② The channel from which the outbound adapter receives messages for performing the desired operation.

③ The fully qualified name of the entity class for the JPA Operation. The `entity-class`, `query`, and `named-query` attributes are mutually exclusive. Optional.

④ An instance of `javax.persistence.EntityManager` used to perform the JPA operations. Optional.

⑤ An instance of `javax.persistence.EntityManagerFactory` used to obtain an instance of `javax.persistence.EntityManager`, which performs the JPA operations. Optional.

⑥ An implementation of `org.springframework.integration.jpa.core.JpaOperations` used to perform the JPA operations. We recommend not providing an implementation of your own but using the default `org.springframework.integration.jpa.core.DefaultJpaOperations` implementation. You can use any one of the `entity-manager`, `entity-manager-factory`, or `jpa-operations` attributes. Optional.

⑦ The JPA QL to be executed by this adapter. Optional.

⑧ The named query that needs to be executed by this adapter. Optional.

⑨ The native query to be executed by this adapter. You can use any one of the `jpa-query`, `named-query`, or `native-query` attributes. Optional.

⑩ The order for this consumer when multiple consumers are registered, thereby managing load-balancing and failover. It defaults to `Ordered.LOWEST_PRECEDENCE`. Optional.

⑪ An instance of `o.s.i.jpa.support.parametersource.ParameterSourceFactory` used to get an instance of `o.s.i.jpa.support.parametersource.ParameterSource`, which is used to resolve the values of the parameters in the query. Ignored if you perform operations by using a JPA entity. The `parameter` sub-elements are mutually exclusive with the `parameter-source-factory` attribute and must be configured on the provided `ParameterSourceFactory`. Optional.

⑫ Accepts one of the following: `PERSIST`, `MERGE`, or `DELETE`. Indicates the operation that the adapter needs to perform. Relevant only if you use an entity for JPA operations. Ignored if you provide JPA QL, a named query, or a native query. It defaults to `MERGE`. Optional. As of Spring Integration 3.0, payloads to persist or merge can also be of type `java.lang.Iterable`. In that case, each object returned by the `Iterable` is treated as an entity and persisted or merged by using the underlying `EntityManager`. Null values returned by the iterator are ignored.

⑬ Set this value to `true` if you want to flush the persistence context immediately after persist, merge, or delete operations and do not want to rely on the `flushMode` of the `EntityManager`. It defaults to `false`. Applies only if you did not specify the `flush-size` attribute. If this attribute is set to `true`, `flush-size` is implicitly set to `1`, if no other value configured it.

⑭ Set this attribute to a value greater than '0' if you want to flush the persistence context immediately after persist, merge or delete operations and do not want to rely on the the `flushMode` of the `EntityManager`. The default value is set to `0`, which means "'no flush'". This attribute is geared towards messages with `Iterable` payloads. For instance, if `flush-size` is set to `3`, then `entityManager.flush()` is called after every third entity. Furthermore, `entityManager.flush()` is called once more after the entire loop. If the 'flush-size' attribute is specified with a value greater than '0', you need not configure the `flush` attribute.

⑮ Set this value to 'true' if you want to clear the persistence context immediately after each flush operation. The attribute's value is applied only if the `flush` attribute is set to `true` or if the `flush-size` attribute is set to a value greater than `0`.

⑯ If set to `true`, the payload of the message is used as a source for parameters. If set to `false`, however, the entire `Message` is available as a source for parameters. Optional.

⑰ Defines the transaction management attributes and the reference to the transaction manager to be used by the JPA adapter. Optional.

⑱ One or more `parameter` attributes — one for each parameter used in the query. The value or expression is evaluated to compute the value of the parameter. Optional.

23.6.6. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the outbound adapter with Java:

```
@SpringBootApplication
@EntityScan(basePackageClasses = StudentDomain.class)
@IntegrationComponentScan
public class JpaJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(JpaJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Autowired
    private EntityManagerFactory entityManagerFactory;

    @MessagingGateway
    interface JpaGateway {

       @Gateway(requestChannel = "jpaPersistChannel")
       @Transactional
       void persistStudent(StudentDomain payload);

    }

    @Bean
    public JpaExecutor jpaExecutor() {
        JpaExecutor executor = new JpaExecutor(this.entityManagerFactory);
        jpaExecutor.setEntityClass(StudentDomain.class);
        jpaExecutor.setPersistMode(PersistMode.PERSIST);
        return executor;
    }

    @Bean
    @ServiceActivator(channel = "jpaPersistChannel")
    public MessageHandler jpaOutbound() {
        JpaOutboundGateway adapter = new JpaOutboundGateway(jpaExecutor());
        adapter.setProducesReply(false);
        return adapter;
    }

}
```

23.6.7. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the outbound adapter with the Java DSL:

```
@SpringBootApplication
@EntityScan(basePackageClasses = StudentDomain.class)
public class JpaJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(JpaJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Autowired
    private EntityManagerFactory entityManagerFactory;

    @Bean
    public IntegrationFlow outboundAdapterFlow() {
        return f -> f
                .handle(Jpa.outboundAdapter(this.entityManagerFactory)
                                .entityClass(StudentDomain.class)
                                .persistMode(PersistMode.PERSIST),
                        e -> e.transactional());
    }

}
```

23.7. Outbound Gateways

The JPA inbound channel adapter lets you poll a database to retrieve one or more JPA entities. The retrieved data is consequently used to start a Spring Integration flow that uses the retrieved data as message payload.

Additionally, you can use JPA outbound channel adapters at the end of your flow in order to persist data, essentially terminating the flow at the end of the persistence operation.

However, how can you execute JPA persistence operations in the middle of a flow? For example, you may have business data that you are processing in your Spring Integration message flow and that you would like to persist, yet you still need to use other components further downstream. Alternatively, instead of polling the database using a poller, you need to execute JPQL queries and actively retrieve data, which is then processed in subsequent components within your flow.

This is where JPA Outbound Gateways come into play. They give you the ability to persist data as well as retrieving data. To facilitate these uses, Spring Integration provides two types of JPA outbound gateways:

- Updating outbound gateway
- Retrieving outbound gateway

Whenever the outbound gateway is used to perform an action that saves, updates, or solely deletes

some records in the database, you need to use an updating outbound gateway. If, for example, you use an `entity` to persist it, a merged and persisted entity is returned as a result. In other cases, the number of records affected (updated or deleted) is returned instead.

When retrieving (selecting) data from the database, we use a retrieving outbound gateway. With a retrieving outbound gateway, we can use JPQL, Named Queries (native or JPQL-based), or Native Queries (SQL) for selecting the data and retrieving the results.

An updating outbound gateway is functionally similar to an outbound channel adapter, except that an updating outbound gateway sends a result to the gateway's reply channel after performing the JPA operation.

A retrieving outbound gateway is similar to an inbound channel adapter.

We recommend you first read the Outbound Channel Adapter section and the Inbound Channel Adapter sections earlier in this chapter, as most of the common concepts are explained there.

This similarity was the main factor to use the central `JpaExecutor` class to unify common functionality as much as possible.

Common for all JPA outbound gateways and similar to the `outbound-channel-adapter`, we can use for performing various JPA operations:

- Entity classes
- JPA Query Language (JPQL)
- Native query
- Named query

For configuration examples see JPA Outbound Gateway Samples.

23.7.1. Common Configuration Parameters

JPA Outbound Gateways always have access to the Spring Integration `Message` as input. Consequently, the following parameters is available:

`parameter-source-factory`

An instance of `o.s.i.jpa.support.parametersource.ParameterSourceFactory` used to get an instance of `o.s.i.jpa.support.parametersource.ParameterSource`. The `ParameterSource` is used to resolve the values of the parameters provided in the query. If you perform operations by using a JPA entity, the `parameter-source-factory` attribute is ignored. The `parameter` sub-elements are mutually exclusive with the `parameter-source-factory` and they have to be configured on the provided `ParameterSourceFactory`. Optional.

`use-payload-as-parameter-source`

If set to `true`, the payload of the `Message` is used as a source for parameters. If set to `false`, the entire `Message` is available as a source for parameters. If no JPA Parameters are passed in, this property defaults to `true`. This means that, if you use a default

`BeanPropertyParameterSourceFactory`, the bean properties of the payload are used as a source for parameter values for the JPA query. However, if JPA Parameters are passed in, this property, by default, evaluates to `false`. The reason is that JPA Parameters let you provide SpEL Expressions. Therefore, it is highly beneficial to have access to the entire `Message`, including the headers. Optional.

23.7.2. Updating Outbound Gateway

The following listing shows all the attributes that you can set on an updating-outbound-gateway and describes the key attributes:

```
<int-jpa:updating-outbound-gateway request-channel=""  ①
    auto-startup="true"
    entity-class=""
    entity-manager=""
    entity-manager-factory=""
    id=""
    jpa-operations=""
    jpa-query=""
    named-query=""
    native-query=""
    order=""
    parameter-source-factory=""
    persist-mode="MERGE"
    reply-channel=""  ②
    reply-timeout=""  ③
    use-payload-as-parameter-source="true">

    <int:poller/>
    <int-jpa:transactional/>

    <int-jpa:parameter name="" type="" value=""/>
    <int-jpa:parameter name="" expression=""/>
</int-jpa:updating-outbound-gateway>
```

① The channel from which the outbound gateway receives messages for performing the desired operation. This attribute is similar to the `channel` attribute of the `outbound-channel-adapter`. Optional.

② The channel to which the gateway send the response after performing the required JPA operation. If this attribute is not defined, the request message must have a `replyChannel` header. Optional.

③ Specifies the time the gateway waits to send the result to the reply channel. Only applies when the reply channel itself might block the send operation (for example, a bounded `QueueChannel` that is currently full). By default, the gateway waits indefinitely. The value is specified in milliseconds. Optional.

The remaining attributes are described earlier in this chapter. See Configuration Parameter

Reference and Configuration Parameter Reference.

23.7.3. Configuring with Java Configuration

The following Spring Boot application shows an example of how configure the outbound adapter with Java:

```
@SpringBootApplication
@EntityScan(basePackageClasses = StudentDomain.class)
@IntegrationComponentScan
public class JpaJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(JpaJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Autowired
    private EntityManagerFactory entityManagerFactory;

    @MessagingGateway
    interface JpaGateway {

       @Gateway(requestChannel = "jpaUpdateChannel")
       @Transactional
       void updateStudent(StudentDomain payload);

    }

    @Bean
    @ServiceActivator(channel = "jpaUpdateChannel")
    public MessageHandler jpaOutbound() {
        JpaOutboundGateway adapter =
                new JpaOutboundGateway(new JpaExecutor(this.entityManagerFactory));
        adapter.setOutputChannelName("updateResults");
        return adapter;
    }

}
```

23.7.4. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the outbound adapter using the Java DSL:

```
@SpringBootApplication
@EntityScan(basePackageClasses = StudentDomain.class)
public class JpaJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(JpaJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Autowired
    private EntityManagerFactory entityManagerFactory;

    @Bean
    public IntegrationFlow updatingGatewayFlow() {
        return f -> f
                .handle(Jpa.updatingGateway(this.entityManagerFactory),
                        e -> e.transactional(true))
                .channel(c -> c.queue("updateResults"));
    }

}
```

23.7.5. Retrieving Outbound Gateway

The following example shows all the attributes that you can set on a retrieving outbound gateway and describes the key attributes:

```
<int-jpa:retrieving-outbound-gateway request-channel=""
    auto-startup="true"
    delete-after-poll="false"
    delete-in-batch="false"
    entity-class=""
    id-expression=""                ①
    entity-manager=""
    entity-manager-factory=""
    expect-single-result="false"    ②
    id=""
    jpa-operations=""
    jpa-query=""
    max-results=""                  ③
    max-results-expression=""       ④
    first-result=""                 ⑤
    first-result-expression=""      ⑥
    named-query=""
    native-query=""
    order=""
    parameter-source-factory=""
    reply-channel=""
    reply-timeout=""
    use-payload-as-parameter-source="true">
    <int:poller></int:poller>
    <int-jpa:transactional/>

    <int-jpa:parameter name="" type="" value=""/>
    <int-jpa:parameter name="" expression=""/>
</int-jpa:retrieving-outbound-gateway>
```

① (Since Spring Integration 4.0) The SpEL expression that determines the `primaryKey` value for `EntityManager.find(Class entityClass, Object primaryKey)` method against the `requestMessage` as the root object of evaluation context. The `entityClass` argument is determined from the `entity-class` attribute, if present. Otherwise, it is determined from the `payload` class. All other attributes are disallowed if you use `id-expression`. Optional.

② A boolean flag indicating whether the select operation is expected to return a single result or a `List` of results. If this flag is set to `true`, a single entity is sent as the payload of the message. If multiple entities are returned, an exception is thrown. If `false`, the `List` of entities is sent as the payload of the message. It defaults to `false`. Optional.

③ This non-zero, non-negative integer value tells the adapter not to select more than the specified number of rows on execution of the select operation. By default, if this attribute is not set, all the possible records are selected by given query. This attribute is mutually exclusive with `max-results-expression`. Optional.

④ An expression that can be used to find the maximum number of results in a result set. It is mutually exclusive with `max-results`. Optional.

⑤ This non-zero, non-negative integer value tells the adapter the first record from which

results are to be retrieved. This attribute is mutually exclusive with `first-result-expression`. Version 3.0 introduced this attribute. Optional.

⑥ This expression is evaluated against the message, to find the position of the first record in the result set. This attribute is mutually exclusive to `first-result`. Version 3.0 introduced this attribute. Optional.

The remaining attributes are described earlier in this chapter. See Configuration Parameter Reference and Configuration Parameter Reference.

23.7.6. Configuring with Java Configuration

The following Spring Boot application shows an example of how configure the outbound adapter with Java:

```
@SpringBootApplication
@EntityScan(basePackageClasses = StudentDomain.class)
public class JpaJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(JpaJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Autowired
    private EntityManagerFactory entityManagerFactory;


    @Bean
    public JpaExecutor jpaExecutor() {
        JpaExecutor executor = new JpaExecutor(this.entityManagerFactory);
        jpaExecutor.setJpaQuery("from Student s where s.id = :id");
        executor.setJpaParameters(Collections.singletonList(new JpaParameter("id",
null, "payload")));
        jpaExecutor.setExpectSingleResult(true);
        return executor;
    }

    @Bean
    @ServiceActivator(channel = "jpaRetrievingChannel")
    public MessageHandler jpaOutbound() {
        JpaOutboundGateway adapter = new JpaOutboundGateway(jpaExecutor());
        adapter.setOutputChannelName("retrieveResults");
        adapter.setGatewayType(OutboundGatewayType.RETRIEVING);
        return adapter;
    }

}
```

23.7.7. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the outbound adapter with the Java DSL:

```
@SpringBootApplication
@EntityScan(basePackageClasses = StudentDomain.class)
public class JpaJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(JpaJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Autowired
    private EntityManagerFactory entityManagerFactory;

    @Bean
    public IntegrationFlow retrievingGatewayFlow() {
        return f -> f
                .handle(Jpa.retrievingGateway(this.entityManagerFactory)
                       .jpaQuery("from Student s where s.id = :id")
                       .expectSingleResult(true)
                       .parameterExpression("id", "payload"))
                .channel(c -> c.queue("retrieveResults"));
    }

}
```

When you choose to delete entities upon retrieval and you have retrieved a collection of entities, by default, entities are deleted on a per-entity basis. This may cause performance issues.

Alternatively, you can set attribute `deleteInBatch` to `true`, which performs a batch delete. However, the limitation of doing so is that cascading deletes are not supported.

JSR 317: Java™ Persistence 2.0 states in chapter 4.10, “Bulk Update and Delete Operations” that:

“A delete operation only applies to entities of the specified class and its subclasses. It does not cascade to related entities.”

For more information, see JSR 317: Java™ Persistence 2.0

23.7.8. JPA Outbound Gateway Samples

This section contains various examples of using the updating outbound gateway and the retrieving outbound gateway:

Update by Using an Entity Class

In the following example, an updating outbound gateway is persisted by using the `org.springframework.integration.jpa.test.entity.Student` entity class as a JPA defining parameter:

```
<int-jpa:updating-outbound-gateway request-channel="entityRequestChannel"  ①
    reply-channel="entityResponseChannel"  ②
    entity-class="org.springframework.integration.jpa.test.entity.Student"
    entity-manager="em"/>
```

① This is the request channel for the outbound gateway. It is similar to the `channel` attribute of the `outbound-channel-adapter`.

② This is where a gateway differs from an outbound adapter. This is the channel over which the reply from the JPA operation is received. If, however, you are not interested in the reply received and want only to perform the operation, using a JPA `outbound-channel-adapter` is the appropriate choice. In this example, where we use an entity class, the reply is the entity object that was created or merged as a result of the JPA operation.

Update using JPQL

The following example updates an entity by using the Java Persistence Query Language (JPQL), which mandates using an updating outbound gateway:

```
<int-jpa:updating-outbound-gateway request-channel="jpaqlRequestChannel"
  reply-channel="jpaqlResponseChannel"
  jpa-query="update Student s set s.lastName = :lastName where s.rollNumber =
:rollNumber"  ①
  entity-manager="em">
    <int-jpa:parameter name="lastName" expression="payload"/>
    <int-jpa:parameter name="rollNumber" expression="headers['rollNumber']"/>
</int-jpa:updating-outbound-gateway>
```

① The JPQL query that the gateway executes. Since we used updating outbound gateway, only `update` and `delete` JPQL queries would be sensible choices.

When you send a message with a `String` payload that also contains a header called `rollNumber` with a `long` value, the last name of the student with the specified roll number is updated to the value in the message payload. When using an updating gateway, the return value is always an integer value, which denotes the number of records affected by execution of the JPA QL.

Retrieving an Entity using JPQL

The following example uses a retrieving outbound gateway and JPQL to retrieve (select) one or more entities from the database:

```
<int-jpa:retrieving-outbound-gateway request-channel="retrievingGatewayReqChannel"
    reply-channel="retrievingGatewayReplyChannel"
    jpa-query="select s from Student s where s.firstName = :firstName and
s.lastName = :lastName"
    entity-manager="em">
    <int-jpa:parameter name="firstName" expression="payload"/>
    <int-jpa:parameter name="lastName" expression="headers['lastName']"/>
</int-jpa:outbound-gateway>
```

Retrieving an Entity by Using `id-expression`

The following example uses a retrieving outbound gateway with `id-expression` to retrieve (find) one and only one entity from the database: The `primaryKey` is the result of `id-expression` evaluation. The `entityClass` is a class of Message `payload`.

```
<int-jpa:retrieving-outbound-gateway
    request-channel="retrievingGatewayReqChannel"
    reply-channel="retrievingGatewayReplyChannel"
    id-expression="payload.id"
    entity-manager="em"/>
```

Update using a Named Query

Using a named query is basically the same as using a JPQL query directly. The difference is that the `named-query` attribute is used instead, as the following example shows:

```
<int-jpa:updating-outbound-gateway request-channel="namedQueryRequestChannel"
    reply-channel="namedQueryResponseChannel"
    named-query="updateStudentByRollNumber"
    entity-manager="em">
    <int-jpa:parameter name="lastName" expression="payload"/>
    <int-jpa:parameter name="rollNumber" expression="headers['rollNumber']"/>
</int-jpa:outbound-gateway>
```

You can find a complete sample application that uses Spring Integration's JPA adapter here.

Chapter 24. JMS Support

Spring Integration provides channel adapters for receiving and sending JMS messages.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-jms</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-jms:5.3.0.RC1"
```

The `javax.jms:javax.jms-api` must be added explicitly via some JMS vendor-specific implementation, e.g. Apache ActiveMQ.

There are actually two JMS-based inbound Channel Adapters. The first uses Spring's `JmsTemplate` to receive based on a polling period. The second is "message-driven" and relies on a Spring `MessageListener` container. The outbound channel adapter uses the `JmsTemplate` to convert and send a JMS message on demand.

By using `JmsTemplate` and the `MessageListener` container, Spring Integration relies on Spring's JMS support. This is important to understand, since most of the attributes exposed on these adapters configure the underlying `JmsTemplate` and `MessageListener` container. For more details about `JmsTemplate` and the `MessageListener` container, see the Spring JMS documentation.

Whereas the JMS channel adapters are intended for unidirectional messaging (send-only or receive-only), Spring Integration also provides inbound and outbound JMS Gateways for request and reply operations. The inbound gateway relies on one of Spring's `MessageListener` container implementations for message-driven reception. It is also capable of sending a return value to the `reply-to` destination, as provided by the received message. The outbound gateway sends a JMS message to a `request-destination` (or `request-destination-name` or `request-destination-expression`) and then receives a reply message. You can explicitly configure the `reply-destination` reference (or `reply-destination-name` or `reply-destination-expression`). Otherwise, the outbound gateway uses a JMS TemporaryQueue.

Prior to Spring Integration 2.2, if necessary, a `TemporaryQueue` was created (and removed) for each request or reply. Beginning with Spring Integration 2.2, you can configure the outbound gateway to use a `MessageListener` container to receive replies instead of directly using a new (or cached) `Consumer` to receive the reply for each request. When so configured, and no explicit reply destination is provided, a single `TemporaryQueue` is used for each gateway instead of one for each request.

24.1. Inbound Channel Adapter

The inbound channel adapter requires a reference to either a single `JmsTemplate` instance or both a `ConnectionFactory` and a `Destination` (you can provide a 'destinationName' in place of the 'destination' reference). The following example defines an inbound channel adapter with a `Destination` reference:

```
<int-jms:inbound-channel-adapter id="jmsIn" destination="inQueue" channel=
"exampleChannel">
    <int:poller fixed-rate="30000"/>
</int-jms:inbound-channel-adapter>
```

Notice from the preceding configuration that the `inbound-channel-adapter` is a polling consumer. That means that it invokes `receive()` when triggered. You should use this should only in situations where polling is done relatively infrequently and timeliness is not important. For all other situations (a vast majority of JMS-based use-cases), the `message-driven-channel-adapter` (described later) is a better option.

By default, all of the JMS adapters that require a reference to the `ConnectionFactory` automatically look for a bean named `jmsConnectionFactory`. That is why you do not see a `connection-factory` attribute in many of the examples. However, if your JMS `ConnectionFactory` has a different bean name, you need to provide that attribute.

If `extract-payload` is set to `true` (the default), the received JMS Message is passed through the `MessageConverter`. When relying on the default `SimpleMessageConverter`, this means that the resulting Spring Integration Message has the JMS message's body as its payload. A JMS `TextMessage` produces a string-based payload, a JMS `BytesMessage` produces a byte array payload, and the serializable instance of a JMS `ObjectMessage` becomes the Spring Integration message's payload. If you prefer to have the raw JMS message as the Spring Integration message's payload, set `extract-payload` to `false`, as the following example shows:

```
<int-jms:inbound-channel-adapter id="jmsIn"
    destination="inQueue"
    channel="exampleChannel"
    extract-payload="false"/>
    <int:poller fixed-rate="30000"/>
</int-jms:inbound-channel-adapter>
```

Starting with version 5.0.8, a default value of the `receive-timeout` is `-1` (no wait) for the `org.springframework.jms.connection.CachingConnectionFactory` and `cacheConsumers`, otherwise it is 1 second.

24.1.1. Transactions

Starting with version 4.0, the inbound channel adapter supports the `session-transacted` attribute. In earlier versions, you had to inject a `JmsTemplate` with `sessionTransacted` set to `true`. (The adapter did let you set the `acknowledge` attribute to `transacted`, but this was incorrect and did not work).

Note, however, that setting `session-transacted` to `true` has little value, because the transaction is committed immediately after the `receive()` operation and before the message is sent to the `channel`.

If you want the entire flow to be transactional (for example, if there is a downstream outbound channel adapter), you must use a `transactional` poller with a `JmsTransactionManager`. Alternatively, consider using a `jms-message-driven-channel-adapter` with `acknowledge` set to `transacted` (the default).

24.2. Message-driven Channel Adapter

The `message-driven-channel-adapter` requires a reference to either an instance of a Spring `MessageListener` container (any subclass of `AbstractMessageListenerContainer`) or both `ConnectionFactory` and `Destination` (a 'destinationName' can be provided in place of the 'destination' reference). The following example defines a message-driven channel adapter with a `Destination` reference:

```
<int-jms:message-driven-channel-adapter id="jmsIn" destination="inQueue" channel=
"exampleChannel"/>
```

The message-driven adapter also accepts several properties that pertain to the `MessageListener` container. These values are considered only if you do not provide a `container` reference. In that case, an instance of `DefaultMessageListenerContainer` is created and configured based on these properties. For example, you can specify the `transaction-manager` reference, the `concurrent-consumers` value, and several other property references and values. See the Javadoc and Spring Integration's JMS schema (`spring-integration-jms.xsd`) for more details.

If you have a custom listener container implementation (usually a subclass of `DefaultMessageListenerContainer`), you can either provide a reference to an instance of it by using the `container` attribute or provide its fully qualified class name by using the `container-class` attribute. In that case, the attributes on the adapter are transferred to an instance of your custom container.

You can't use the Spring JMS namespace element `<jms:listener-container/>` to configure a container reference for the `<int-jms:message-driven-channel-adapter>` since that element doesn't actually reference a container. Each `<jms:listener/>` sub-element gets its own `DefaultMessageListenerContainer` (with shared attributes defined on the parent `<jms:listener-container/>` element). You can give each listener sub-element an `id`, and use that to inject into the channel adapter, however, the `<jms:/>` namespace requires a real listener. Since, for Spring Integration, the adapter itself needs to configure the listener, the configured listener will be overwritten. If you go this route, you will see a warning for each adapter.

It is recommended to configure a regular `<bean>` for the `DefaultMessageListenerContainer` and use it as a reference in the channel adapter.

Starting with version 4.2, the default `acknowledge` mode is `transacted`, unless you provide an external container. In that case, you should configure the container as needed. We recommend using `transacted` with the `DefaultMessageListenerContainer` to avoid message loss.

The 'extract-payload' property has the same effect, and its default value is 'true'. The `poller` element is not applicable for a message-driven channel adapter, as it is actively invoked. For most scenarios, the message-driven approach is better, since the messages are passed along to the `MessageChannel` as soon as they are received from the underlying JMS consumer.

Finally, the `<message-driven-channel-adapter>` element also accepts the 'error-channel' attribute. This provides the same basic functionality, as described in Enter the `GatewayProxyFactoryBean`. The following example shows how to set an error channel on a message-driven channel adapter:

```
<int-jms:message-driven-channel-adapter id="jmsIn" destination="inQueue"
    channel="exampleChannel"
    error-channel="exampleErrorChannel"/>
```

When comparing the preceding example to the generic gateway configuration or the JMS 'inbound-gateway' that we discuss later, the key difference is that we are in a one-way flow, since this is a 'channel-adapter', not a gateway. Therefore, the flow downstream from the 'error-channel' should also be one-way. For example, it could send to a logging handler or it could connect to a different JMS `<outbound-channel-adapter>` element.

When consuming from topics, set the `pub-sub-domain` attribute to true. Set `subscription-durable` to `true` for a durable subscription or `subscription-shared` for a shared subscription (which requires a JMS 2.0 broker and has been available since version 4.2). Use `subscription-name` to name the subscription.

Starting with version 5.1, when the endpoint is stopped while the application remains running, the underlying listener container is shut down, closing its shared connection and consumers. Previously, the connection and consumers remained open. To revert to the previous behavior, set

the shutdownContainerOnStop on the JmsMessageDrivenEndpoint to false.

24.2.1. Inbound Conversion Errors

Starting with version 4.2, the 'error-channel' is used for the conversion errors, too. Previously, if a JMS <message-driven-channel-adapter/> or <inbound-gateway/> could not deliver a message due to a conversion error, an exception would be thrown back to the container. If the container is configured to use transactions, the message is rolled back and redelivered repeatedly. The conversion process occurs before and during message construction so that such errors are not sent to the 'error-channel'. Now such conversion exceptions result in an ErrorMessage being sent to the 'error-channel', with the exception as the payload. If you wish the transaction to roll back and you have an 'error-channel' defined, the integration flow on the 'error-channel' must re-throw the exception (or another exception). If the error flow does not throw an exception, the transaction is committed and the message is removed. If no 'error-channel' is defined, the exception is thrown back to the container, as before.

24.3. Outbound Channel Adapter

The JmsSendingMessageHandler implements the MessageHandler interface and is capable of converting Spring Integration Messages to JMS messages and then sending to a JMS destination. It requires either a jmsTemplate reference or both jmsConnectionFactory and destination references (destinationName may be provided in place of destination). As with the inbound channel adapter, the easiest way to configure this adapter is with the namespace support. The following configuration produces an adapter that receives Spring Integration messages from the exampleChannel, converts those into JMS messages, and sends them to the JMS destination reference whose bean name is outQueue:

```
<int-jms:outbound-channel-adapter id="jmsOut" destination="outQueue" channel=
"exampleChannel"/>
```

As with the inbound channel adapters, there is an 'extract-payload' property. However, the meaning is reversed for the outbound adapter. Rather than applying to the JMS message, the boolean property applies to the Spring Integration message payload. In other words, the decision is whether to pass the Spring Integration message itself as the JMS message body or to pass the Spring Integration message payload as the JMS message body. The default value is 'true'. Therefore, if you pass a Spring Integration message whose payload is a String, a JMS TextMessage is created. If, on the other hand, you want to send the actual Spring Integration message to another system over JMS, set it to 'false'.

Regardless of the boolean value for payload extraction, the Spring Integration MessageHeaders map to JMS properties, as long as you rely on the default converter or provide a reference to another instance of MessageConverter. (The same holds true for 'inbound' adapters, except that, in those cases, the JMS properties map to Spring Integration MessageHeaders).

Starting with version 5.1, the `<int-jms:outbound-channel-adapter>` (`JmsSendingMessageHandler`) can be configured with the `deliveryModeExpression` and `timeToLiveExpression` properties to evaluate an appropriate QoS values for JMS message to send at runtime against request Spring `Message`. The new `setMapInboundDeliveryMode(true)` and `setMapInboundExpiration(true)` options of the `DefaultJmsHeaderMapper` may facilitate as a source of the information for the dynamic `deliveryMode` and `timeToLive` from message headers:

```
<int-jms:outbound-channel-adapter delivery-mode-expression=
"headers.jms_deliveryMode"
                        time-to-live-expression="headers.jms_expiration -
T(System).currentTimeMillis()"/>
```

24.3.1. Transactions

Starting with version 4.0, the outbound channel adapter supports the `session-transacted` attribute. In earlier versions, you had to inject a `JmsTemplate` with `sessionTransacted` set to `true`. The attribute now sets the property on the built-in default `JmsTemplate`. If a transaction exists (perhaps from an upstream `message-driven-channel-adapter`), the send operation is performed within the same transaction. Otherwise, a new transaction is started.

24.4. Inbound Gateway

Spring Integration's message-driven JMS inbound-gateway delegates to a `MessageListener` container, supports dynamically adjusting concurrent consumers, and can also handle replies. The inbound gateway requires references to a `ConnectionFactory` and a request `Destination` (or 'requestDestinationName'). The following example defines a JMS `inbound-gateway` that receives from the JMS queue referenced by the bean id, `inQueue`, and sends to the Spring Integration channel named `exampleChannel`:

```
<int-jms:inbound-gateway id="jmsInGateway"
    request-destination="inQueue"
    request-channel="exampleChannel"/>
```

Since the gateways provide request-reply behavior instead of unidirectional send or receive behavior, they also have two distinct properties for "payload extraction" (as discussed earlier for the channel adapters' 'extract-payload' setting). For an inbound gateway, the 'extract-request-payload' property determines whether the received JMS Message body is extracted. If 'false', the JMS message itself becomes the Spring Integration message payload. The default is 'true'.

Similarly, for an inbound-gateway, the 'extract-reply-payload' property applies to the Spring Integration message that is to be converted into a reply JMS Message. If you want to pass the whole Spring Integration message (as the body of a JMS ObjectMessage), set value this to 'false'. By default, it is also 'true' that the Spring Integration message payload is converted into a JMS Message (for

example, a `String` payload becomes a JMS TextMessage).

As with anything else, gateway invocation might result in error. By default, a producer is not notified of the errors that might have occurred on the consumer side and times out waiting for the reply. However, there might be times when you want to communicate an error condition back to the consumer (in other words, you might want to treat the exception as a valid reply by mapping it to a message). To accomplish this, JMS inbound gateway provides support for a message channel to which errors can be sent for processing, potentially resulting in a reply message payload that conforms to some contract that defines what a caller may expect as an "error" reply. You can use the error-channel attribute to configure such a channel, as the following example shows:

```
<int-jms:inbound-gateway request-destination="requestQueue"
          request-channel="jmsInputChannel"
          error-channel="errorTransformationChannel"/>

<int:transformer input-channel="exceptionTransformationChannel"
        ref="exceptionTransformer" method="createErrorResponse"/>
```

You might notice that this example looks very similar to that included within Enter the `GatewayProxyFactoryBean`. The same idea applies here: The `exceptionTransformer` could be a POJO that creates error-response objects, you could reference the `nullChannel` to suppress the errors, or you could leave 'error-channel' out to let the exception propagate.

See Inbound Conversion Errors.

When consuming from topics, set the `pub-sub-domain` attribute to true. Set `subscription-durable` to `true` for a durable subscription or `subscription-shared` for a shared subscription (requires a JMS 2.0 broker and has been available since version 4.2). Use `subscription-name` to name the subscription.

Starting with version 4.2, the default `acknowledge` mode is `transacted`, unless an external container is provided. In that case, you should configure the container as needed. We recommend that you use `transacted` with the `DefaultMessageListenerContainer` to avoid message loss.

Starting with version 5.1, when the endpoint is stopped while the application remains running, the underlying listener container is shut down, closing its shared connection and consumers. Previously, the connection and consumers remained open. To revert to the previous behavior, set the `shutdownContainerOnStop` on the `JmsInboundGateway` to `false`.

24.5. Outbound Gateway

The outbound gateway creates JMS messages from Spring Integration messages and sends them to a 'request-destination'. It thens handle the JMS reply message either by using a selector to receive from the 'reply-destination' that you configure or, if no 'reply-destination' is provided, by creating JMS `TemporaryQueue` instances.

Using a `reply-destination` (or `reply-destination-name`) together with a `CachingConnectionFactory` that has cacheConsumers set to `true` can cause out-of-memory conditions. This is because each request gets a new consumer with a new selector (selecting on the `correlation-key` value or, when there is no `correlation-key`, on the sent JMSMessageID). Given that these selectors are unique, they remain in the cache (unused) after the current request completes.

If you specify a reply destination, you are advised to not use cached consumers. Alternatively, consider using a `<reply-listener/>` as described below.

The following example shows how to configure an outbound gateway:

```
<int-jms:outbound-gateway id="jmsOutGateway"
    request-destination="outQueue"
    request-channel="outboundJmsRequests"
    reply-channel="jmsReplies"/>
```

The 'outbound-gateway' payload extraction properties are inversely related to those of the 'inbound-gateway' (see the earlier discussion). That means that the 'extract-request-payload' property value applies to the Spring Integration message being converted into a JMS message to be sent as a request. The 'extract-reply-payload' property value applies to the JMS message received as a reply and is then converted into a Spring Integration message to be subsequently sent to the 'reply-channel', as shown in the preceding configuration example.

24.5.1. Using a `<reply-listener/>`

Spring Integration 2.2 introduced an alternative technique for handling replies. If you add a `<reply-listener/>` child element to the gateway instead of creating a consumer for each reply, a `MessageListener` container is used to receive the replies and hand them over to the requesting thread. This provides a number of performance benefits as well as alleviating the cached consumer memory utilization problem described in the earlier caution.

When using a `<reply-listener/>` with an outbound gateway that has no `reply-destination`, instead of creating a `TemporaryQueue` for each request, a single `TemporaryQueue` is used. (The gateway creates an additional `TemporaryQueue`, as necessary, if the connection to the broker is lost and recovered).

When using a `correlation-key`, multiple gateways can share the same reply destination, because the listener container uses a selector that is unique to each gateway.

If you specify a reply listener and specify a reply destination (or reply destination name) but provide no correlation key, the gateway logs a warning and falls back to pre-version 2.2 behavior. This is because there is no way to configure a selector in this case. Thus, there is no way to avoid a reply going to a different gateway that might be configured with the same reply destination.

Note that, in this situation, a new consumer is used for each request, and consumers can build up in memory as described in the caution above; therefore cached consumers should not be used in this case.

The following example shows a reply listener with default attributes:

```
<int-jms:outbound-gateway id="jmsOutGateway"
        request-destination="outQueue"
        request-channel="outboundJmsRequests"
        reply-channel="jmsReplies">
    <int-jms:reply-listener />
</int-jms-outbound-gateway>
```

The listener is very lightweight, and we anticipate that, in most cases, you need only a single consumer. However, you can add attributes such as `concurrent-consumers`, `max-concurrent-consumers`, and others. See the schema for a complete list of supported attributes, together with the Spring JMS documentation for their meanings.

24.5.2. Idle Reply Listeners

Starting with version 4.2, you can start the reply listener as needed (and stop it after an idle time) instead of running for the duration of the gateway's lifecycle. This can be useful if you have many gateways in the application context where they are mostly idle. One such situation is a context with many (inactive) partitioned Spring Batch jobs using Spring Integration and JMS for partition distribution. If all the reply listeners are active, the JMS broker has an active consumer for each gateway. By enabling the idle timeout, each consumer exists only while the corresponding batch job is running (and for a short time after it finishes).

See `idle-reply-listener-timeout` in Attribute Reference.

24.5.3. Gateway Reply Correlation

This section describes the mechanisms used for reply correlation (ensuring the originating gateway receives replies to only its requests), depending on how the gateway is configured. See Attribute Reference for complete description of the attributes discussed here.

The following list describes the various scenarios (the numbers are for identification — order does not matter):

1. No `reply-destination*` properties and no `<reply-listener>`

A `TemporaryQueue` is created for each request and deleted when the request is complete (successfully or otherwise). `correlation-key` is irrelevant.

2. A `reply-destination*` property is provided and neither a `<reply-listener/>` nor a `correlation-key` is provided

   The `JMSCorrelationID` equal to the outgoing message IS is used as a message selector for the consumer:

   ```
   messageSelector = "JMSCorrelationID = '" + messageId + "'"
   ```

   The responding system is expected to return the inbound `JMSMessageID` in the reply `JMSCorrelationID`. This is a common pattern and is implemented by the Spring Integration inbound gateway as well as Spring's `MessageListenerAdapter` for message-driven POJOs.

   When you use this configuration, you should not use a topic for replies. The reply may be lost.

3. A `reply-destination*` property is provided, no `<reply-listener/>` is provided, and `correlation-key="JMSCorrelationID"`

   The gateway generates a unique correlation IS and inserts it in the `JMSCorrelationID` header. The message selector is:

   ```
   messageSelector = "JMSCorrelationID = '" + uniqueId + "'"
   ```

   The responding system is expected to return the inbound `JMSCorrelationID` in the reply `JMSCorrelationID`. This is a common pattern and is implemented by the Spring Integration inbound gateway as well as Spring's `MessageListenerAdapter` for message-driven POJOs.

4. A `reply-destination*` property is provided, no `<reply-listener/>` is provided, and `correlation-key="myCorrelationHeader"`

   The gateway generates a unique correlation ID and inserts it in the `myCorrelationHeader` message property. The `correlation-key` can be any user-defined value. The message selector is:

   ```
   messageSelector = "myCorrelationHeader = '" + uniqueId + "'"
   ```

   The responding system is expected to return the inbound `myCorrelationHeader` in the reply `myCorrelationHeader`.

5. A `reply-destination*` property is provided, no `<reply-listener/>` is provided, and `correlation-key="JMSCorrelationID*"` (Note the `*` in the correlation key.)

   The gateway uses the value in the `jms_correlationId` header (if present) from the request message and inserts it in the `JMSCorrelationID` header. The message selector is:

   ```
   messageSelector = "JMSCorrelationID = '" + headers['jms_correlationId'] + "'"
   ```

   The user must ensure this value is unique.

   If the header does not exist, the gateway behaves as in 3.

The responding system is expected to return the inbound `JMSCorrelationID` in the reply `JMSCorrelationID`. This is a common pattern and is implemented by the Spring Integration inbound gateway as well as Spring's `MessageListenerAdapter` for message-driven POJOs.

6. No `reply-destination*` properties is provided, and a `<reply-listener>` is provided

   A temporary queue is created and used for all replies from this gateway instance. No correlation data is needed in the message, but the outgoing `JMSMessageID` is used internally in the gateway to direct the reply to the correct requesting thread.

7. A `reply-destination*` property is provided, a `<reply-listener>` is provided, and no `correlation-key` is provided

   Not allowed.

   The `<reply-listener/>` configuration is ignored, and the gateway behaves as in `2`. A warning log message is written to indicate this situation.

8. A `reply-destination*` property is provided, a `<reply-listener>` is provided, and `correlation-key="JMSCorrelationID"`

   The gateway has a unique correlation ID and inserts it, together with an incrementing value in the `JMSCorrelationID` header (`gatewayId + "_" + ++seq`). The message selector is:

   ```
   messageSelector = "JMSCorrelationID LIKE '" + gatewayId%'"
   ```

   The responding system is expected to return the inbound `JMSCorrelationID` in the reply `JMSCorrelationID`. This is a common pattern and is implemented by the Spring Integration inbound gateway as well as Spring's `MessageListenerAdapter` for message-driven POJOs. Since each gateway has a unique ID, each instance gets only its own replies. The complete correlation data is used to route the reply to the correct requesting thread.

9. A `reply-destination*` property is provided a `<reply-listener/>` is provided, and `correlation-key="myCorrelationHeader"`

   The gateway has a unique correlation ID and inserts it, together with an incrementing value in the `myCorrelationHeader` property (`gatewayId + "_" + ++seq`). The `correlation-key` can be any user-defined value. The message selector is:

   ```
   messageSelector = "myCorrelationHeader LIKE '" + gatewayId%'"
   ```

   The responding system is expected to return the inbound `myCorrelationHeader` in the reply `myCorrelationHeader`. Since each gateway has a unique ID, each instance only gets its own replies. The complete correlation data is used to route the reply to the correct requesting thread.

10. A `reply-destination*` property is provided, a `<reply-listener/>` is provided, and `correlation-key="JMSCorrelationID*"`

    (Note the * in the correlation key)

    Not allowed.

    User-supplied correlation IDs are not permitted with a reply listener. The gateway does not

initialize with this configuration.

24.5.4. Async Gateway

Starting with version 4.3, you can now specify `async="true"` (or `setAsync(true)` in Java) when configuring the outbound gateway.

By default, when a request is sent to the gateway, the requesting thread is suspended until the reply is received. The flow then continues on that thread. If `async` is `true`, the requesting thread is released immediately after the send completes, and the reply is returned (and the flow continues) on the listener container thread. This can be useful when the gateway is invoked on a poller thread. The thread is released and is available for other tasks within the framework.

`async` requires a `<reply-listener/>` (or `setUseReplyContainer(true)` when using Java configuration). It also requires a `correlationKey` (usually `JMSCorrelationID`) to be specified. If either of these conditions are not met, `async` is ignored.

24.5.5. Attribute Reference

The following listing shows all the available attributes for an `outbound-gateway`:

```
<int-jms:outbound-gateway
    connection-factory="connectionFactory" ①
    correlation-key="" ②
    delivery-persistent="" ③
    destination-resolver="" ④
    explicit-qos-enabled="" ⑤
    extract-reply-payload="true" ⑥
    extract-request-payload="true" ⑦
    header-mapper="" ⑧
    message-converter="" ⑨
    priority="" ⑩
    receive-timeout="" ⑪
    reply-channel="" ⑫
    reply-destination="" ⑬
    reply-destination-expression="" ⑭
    reply-destination-name="" ⑮
    reply-pub-sub-domain="" ⑯
    reply-timeout="" ⑰
    request-channel="" ⑱
    request-destination="" ⑲
    request-destination-expression="" ⑳
    request-destination-name=""
    request-pub-sub-domain=""
    time-to-live=""
    requires-reply=""
    idle-reply-listener-timeout=""
    async="">
  <int-jms:reply-listener />
</int-jms:outbound-gateway>
```

① Reference to a `javax.jms.ConnectionFactory`. The default `jmsConnectionFactory`.

② The name of a property that contains correlation data to correlate responses with replies. If omitted, the gateway expects the responding system to return the value of the outbound `JMSMessageID` header in the `JMSCorrelationID` header. If specified, the gateway generates a correlation ID and populates the specified property with it. The responding system must echo back that value in the same property. It can be set to `JMSCorrelationID`, in which case the standard header is used instead of a `String` property to hold the correlation data. When you use a `<reply-container/>`, you must specify the `correlation-key` if you provide an explicit `reply-destination`. Starting with version 4.0.1, this attribute also supports the value `JMSCorrelationID*`, which means that if the outbound message already has a `JMSCorrelationID` (mapped from the `jms_correlationId`) header, it is used instead of generating a new one. Note, the `JMSCorrelationID*` key is not allowed when you use a `<reply-container/>`, because the container needs to set up a message selector during initialization.

You should understand that the gateway has no way to ensure uniqueness, and unexpected side effects can occur if the provided correlation ID is not unique.

③ A boolean value indicating whether the delivery mode should be `DeliveryMode.PERSISTENT` (`true`) or `DeliveryMode.NON_PERSISTENT` (`false`). This setting takes effect only if `explicit-qos-enabled` is `true`.

④ A `DestinationResolver`. The default is a `DynamicDestinationResolver`, which maps the destination name to a queue or topic of that name.

⑤ When set to `true`, it enables the use of quality of service attributes: `priority`, `delivery-mode`, and `time-to-live`.

⑥ When set to `true` (the default), the payload of the Spring Integration reply message is created from the JMS Reply message's body (by using the `MessageConverter`). When set to `false`, the entire JMS message becomes the payload of the Spring Integration message.

⑦ When set to `true` (the default), the payload of the Spring Integration message is converted to a `JMSMessage` (by using the `MessageConverter`). When set to `false`, the entire Spring Integration Message is converted to the `JMSMessage`. In both cases, the Spring Integration message headers are mapped to JMS headers and properties by using the `HeaderMapper`.

⑧ A `HeaderMapper` used to map Spring Integration message headers to and from JMS message headers and properties.

⑨ A reference to a `MessageConverter` for converting between JMS messages and the Spring Integration message payloads (or messages if `extract-request-payload` is `false`). The default is a `SimpleMessageConverter`.

⑩ The default priority of request messages. Overridden by the message priority header, if present. Its range is `0` to `9`. This setting takes effect only if `explicit-qos-enabled` is `true`.

⑪ The time (in milliseconds) to wait for a reply. The default is `5000` (five seconds).

⑫ The channel to which the reply message is sent.

⑬ A reference to a `Destination`, which is set as the `JMSReplyTo` header. At most, only one of `reply-destination`, `reply-destination-expression`, or `reply-destination-name` is allowed. If none is provided, a `TemporaryQueue` is used for replies to this gateway.

⑭ A SpEL expression evaluating to a `Destination`, which will be set as the `JMSReplyTo` header. The expression can result in a `Destination` object or a `String`. It is used by the `DestinationResolver` to resolve the actual `Destination`. At most, only one of `reply-destination`, `reply-destination-expression`, or `reply-destination-name` is allowed. If none is provided, a `TemporaryQueue` is used for replies to this gateway.

⑮ The name of the destination that is set as the JMSReplyTo header. It is used by the `DestinationResolver` to resolve the actual `Destination`. At most, only one of `reply-destination`, `reply-destination-expression`, or `reply-destination-name` is allowed. If none is provided, a `TemporaryQueue` is used for replies to this gateway.

⑯ When set to `true`, it indicates that any reply `Destination` resolved by the `DestinationResolver` should be a `Topic` rather then a `Queue`.

⑰ The time the gateway waits when sending the reply message to the `reply-channel`. This only has an effect if the `reply-channel` can block — such as a `QueueChannel` with a capacity limit that is currently full. The default is infinity.

⑱ The channel on which this gateway receives request messages.

⑲ A reference to a `Destination` to which request messages are sent. One of `reply-destination`, `reply-destination-expression`, or `reply-destination-name` is required. You can use only one of those three attributes.

⑳ A SpEL expression evaluating to a `Destination` to which request messages are sent. The expression can result in a `Destination` object or a `String`. It is used by the `DestinationResolver` to resolve the actual `Destination`. Oneof `reply-destination`, `reply-destination-expression`, or `reply-destination-name` is required. You can use only one of those three attributes.

The name of the destination to which request messages are sent. It is used by the `DestinationResolver` to resolve the actual `Destination`. One of `reply-destination`, `reply-destination-expression`, or `reply-destination-name` is required. You can use only one of those three attributes.

When set to `true`, it indicates that any request `Destination` resolved by the `DestinationResolver` should be a `Topic` rather then a `Queue`.

Specifies the message time to live. This setting takes effect only if `explicit-qos-enabled` is `true`.

Specifies whether this outbound gateway must return a non-null value. By default, this value is `true`, and a `MessageTimeoutException` is thrown when the underlying service does not return a value after the `receive-timeout`. Note that, if the service is never expected to return a reply, it would be better to use a `<int-jms:outbound-channel-adapter/>` instead of a `<int-jms:outbound-gateway/>` with `requires-reply="false"`. With the latter, the sending thread is blocked, waiting for a reply for the `receive-timeout` period.

When you use a `<reply-listener />`, its lifecycle (start and stop) matches that of the gateway by default. When this value is greater than `0`, the container is started on demand (when a request is sent). The container continues to run until at least this time elapses with no requests being received (and until no replies are outstanding). The container is started again on the next request. The stop time is a minimum and may actually be up to 1.5x this value.

See Async Gateway.

When this element is included, replies are received by an asynchronous `MessageListenerContainer` rather than creating a consumer for each reply. This can be more efficient in many cases.

24.6. Mapping Message Headers to and from JMS Message

JMS messages can contain meta-information such as JMS API headers and simple properties. You can map those to and from Spring Integration message headers by using `JmsHeaderMapper`. The JMS API headers are passed to the appropriate setter methods (such as `setJMSReplyTo`), whereas other headers are copied to the general properties of the JMS Message. JMS outbound gateway is bootstrapped with the default implementation of `JmsHeaderMapper`, which will map standard JMS API Headers as well as primitive or `String` message headers. You could also provide a custom header mapper by using the `header-mapper` attribute of inbound and outbound gateways.

Many JMS vendor-specific clients don't allow setting the `deliveryMode`, `priority` and `timeToLive` properties directly on an already created JMS message. They are considered to be QoS properties and therefore have to be propagated to the target `MessageProducer.send(message, deliveryMode, priority, timeToLive)` API. For this reason the `DefaultJmsHeaderMapper` doesn't map appropriate Spring Integration headers (or expression results) into the mentioned JMS message properties. Instead, a `DynamicJmsTemplate` is used by the `JmsSendingMessageHandler` to propagate header values from the request message into the `MessageProducer.send()` API. To enable this feature, you must configure the outbound endpoint with a `DynamicJmsTemplate` with its `explicitQosEnabled` property set to true. The Spring Integration Java DSL configures a `DynamicJmsTemplate` by default but you must still set the `explicitQosEnabled` property.

Since version 4.0, the `JMSPriority` header is mapped to the standard `priority` header for inbound messages. Previously, the `priority` header was only used for outbound messages. To revert to the previous behavior (that is, to not map the inbound priority), set the `mapInboundPriority` property of `DefaultJmsHeaderMapper` to `false`.

Since version 4.3, the `DefaultJmsHeaderMapper` maps the standard `correlationId` header as a message property by invoking its `toString()` method (`correlationId` is often a `UUID`, which is not supported by JMS). On the inbound side, it is mapped as a `String`. This is independent of the `jms_correlationId` header, which is mapped to and from the `JMSCorrelationID` header. The `JMSCorrelationID` is generally used to correlate requests and replies, whereas the `correlationId` is often used to combine related messages into a group (such as with an aggregator or a resequencer).

Starting with version 5.1, the `DefaultJmsHeaderMapper` can be configured for mapping inbound `JMSDeliveryMode` and `JMSExpiration` properties:

```
@Bean
public DefaultJmsHeaderMapper jmsHeaderMapper() {
    DefaultJmsHeaderMapper mapper = new DefaultJmsHeaderMapper();
    mapper.setMapInboundDeliveryMode(true)
    mapper.setMapInboundExpiration(true)
    return mapper;
}
```

These JMS properties are mapped to the `JmsHeaders.DELIVERY_MODE` and `JmsHeaders.EXPIRATION` Spring Message headers respectively.

24.7. Message Conversion, Marshalling, and Unmarshalling

If you need to convert the message, all JMS adapters and gateways let you provide a `MessageConverter` by setting the `message-converter` attribute. To do so, provide the bean name of an instance of `MessageConverter` that is available within the same ApplicationContext. Also, to provide some consistency with marshaller and unmarshaller interfaces, Spring provides `MarshallingMessageConverter`, which you can configure with your own custom marshallers and unmarshallers. The following example shows how to do so

```
<int-jms:inbound-gateway request-destination="requestQueue"
    request-channel="inbound-gateway-channel"
    message-converter="marshallingMessageConverter"/>

<bean id="marshallingMessageConverter"
    class="org.springframework.jms.support.converter.MarshallingMessageConverter">
    <constructor-arg>
        <bean class="org.bar.SampleMarshaller"/>
    </constructor-arg>
    <constructor-arg>
        <bean class="org.bar.SampleUnmarshaller"/>
    </constructor-arg>
</bean>
```

When you provide your own `MessageConverter` instance, it is still wrapped within the `HeaderMappingMessageConverter`. This means that the 'extract-request-payload' and 'extract-reply-payload' properties can affect the actual objects passed to your converter. The `HeaderMappingMessageConverter` itself delegates to a target `MessageConverter` while also mapping the Spring Integration `MessageHeaders` to JMS message properties and back again.

24.8. JMS-backed Message Channels

The channel adapters and gateways featured earlier are all intended for applications that integrate with other external systems. The inbound options assume that some other system is sending JMS messages to the JMS destination, and the outbound options assume that some other system is receiving from the destination. The other system may or may not be a Spring Integration application. Of course, when sending a Spring Integration message instance as the body of the JMS message itself (with 'extract-payload' value set to `false`), it is assumed that the other system is based on Spring Integration. However, that is by no means a requirement. That flexibility is one of the benefits of using a message-based integration option with the abstraction of “channels”(or destinations in the case of JMS).

Sometimes, both the producer and consumer for a given JMS Destination are intended to be part of the same application, running within the same process. You can accomplish this by using a pair of

inbound and outbound channel adapters. The problem with that approach is that you need two adapters, even though, conceptually, the goal is to have a single message channel. A better option is supported as of Spring Integration version 2.0. Now it is possible to define a single “channel” when using the JMS namespace, as the following example shows:

```
<int-jms:channel id="jmsChannel" queue="exampleQueue"/>
```

The channel in the preceding example behaves much like a normal `<channel/>` element from the main Spring Integration namespace. It can be referenced by both the `input-channel` and `output-channel` attributes of any endpoint. The difference is that this channel is backed by a JMS Queue instance named `exampleQueue`. This means that asynchronous messaging is possible between the producing and consuming endpoints. However, unlike the simpler asynchronous message channels created by adding a `<queue/>` element within a non-JMS `<channel/>` element, the messages are not stored in an in-memory queue. Instead, those messages are passed within a JMS message body, and the full power of the underlying JMS provider is then available for that channel. Probably the most common rationale for using this alternative is to take advantage of the persistence made available by the store-and-forward approach of JMS messaging.

If configured properly, the JMS-backed message channel also supports transactions. In other words, a producer would not actually write to a transactional JMS-backed channel if its send operation is part of a transaction that rolls back. Likewise, a consumer would not physically remove a JMS message from the channel if the reception of that message is part of a transaction that rolls back. Note that the producer and consumer transactions are separate in such a scenario. This is significantly different than the propagation of a transactional context across a simple, synchronous `<channel/>` element that has no `<queue/>` child element.

Since the preceding example above references a JMS Queue instance, it acts as a point-to-point channel. If, on the other hand, you need publish-subscribe behavior, you can use a separate element and reference a JMS Topic instead. The following example shows how to do so:

```
<int-jms:publish-subscribe-channel id="jmsChannel" topic="exampleTopic"/>
```

For either type of JMS-backed channel, the name of the destination may be provided instead of a reference, as the following example shows:

```
<int-jms:channel id="jmsQueueChannel" queue-name="exampleQueueName"/>

<jms:publish-subscribe-channel id="jmsTopicChannel" topic-name="exampleTopicName
"/>
```

In the preceding examples, the destination names are resolved by Spring’s default

`DynamicDestinationResolver` implementation, but you could provide any implementation of the `DestinationResolver` interface. Also, the JMS `ConnectionFactory` is a required property of the channel, but, by default, the expected bean name would be `jmsConnectionFactory`. The following example provides both a custom instance for resolution of the JMS destination names and a different name for the `ConnectionFactory`:

```
<int-jms:channel id="jmsChannel" queue-name="exampleQueueName"
    destination-resolver="customDestinationResolver"
    connection-factory="customConnectionFactory"/>
```

For the `<publish-subscribe-channel />`, set the `durable` attribute to `true` for a durable subscription or `subscription-shared` for a shared subscription (requires a JMS 2.0 broker and has been available since version 4.2). Use `subscription` to name the subscription.

24.9. Using JMS Message Selectors

With JMS message selectors, you can filter JMS Messages based on JMS headers as well as JMS properties. For example, if you want to listen to messages whose custom JMS header property, `myHeaderProperty`, equals `something`, you can specify the following expression:

```
myHeaderProperty = 'something'
```

Message selector expressions are a subset of the SQL-92 conditional expression syntax and are defined as part of the Java Message Service specification (Version 1.1, April 12, 2002). Specifically, see chapter "3.8, Message Selection". It contains a detailed explanation of the expressions syntax.

You can specify the JMS message `selector` attribute by using XML namespace configuration for the following Spring Integration JMS components:

- JMS Channel
- JMS Publish Subscribe Channel
- JMS Inbound Channel Adapter
- JMS Inbound Gateway
- JMS Message-driven Channel Adapter

You cannot reference message body values by using JMS Message selectors.

24.10. JMS Samples

To experiment with these JMS adapters, check out the JMS samples available in the Spring Integration Samples Git repository at https://github.com/SpringSource/spring-integration-

samples/tree/master/basic/jms.

That repository includes two samples. One provides inbound and outbound channel adapters, and the other provides inbound and outbound gateways. They are configured to run with an embedded ActiveMQ process, but you can modify the common.xml Spring application context file of each sample to support either a different JMS provider or a standalone ActiveMQ process.

In other words, you can split the configuration so that the inbound and outbound adapters run in separate JVMs. If you have ActiveMQ installed, modify the `brokerURL` property within the common.xml file to use `tcp://localhost:61616` (instead of `vm://localhost`). Both of the samples accept input from stdin and echo back to stdout. Look at the configuration to see how these messages are routed over JMS.

Chapter 25. Mail Support

This section describes how to work with mail messages in Spring Integration.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-mail</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-mail:5.3.0.RC1"
```

The `javax.mail:javax.mail-api` must be included via vendor-specific implementation.

25.1. Mail-sending Channel Adapter

Spring Integration provides support for outbound email with the `MailSendingMessageHandler`. It delegates to a configured instance of Spring's `JavaMailSender`, as the following example shows:

```
 JavaMailSender mailSender = context.getBean("mailSender", JavaMailSender.class);

 MailSendingMessageHandler mailSendingHandler = new MailSendingMessageHandler
(mailSender);
```

`MailSendingMessageHandler` has various mapping strategies that use Spring's `MailMessage` abstraction. If the received message's payload is already a `MailMessage` instance, it is sent directly. Therefore, we generally recommend that you precede this consumer with a transformer for non-trivial `MailMessage` construction requirements. However, Spring Integration supports a few simple message mapping strategies. For example, if the message payload is a byte array, that is mapped to an attachment. For simple text-based emails, you can provide a string-based message payload. In that case, a `MailMessage` is created with that `String` as the text content. If you work with a message payload type whose `toString()` method returns appropriate mail text content, consider adding Spring Integration's `ObjectToStringTransformer` prior to the outbound mail adapter (see the example in Configuring a Transformer with XML for more detail).

You can also configure the outbound `MailMessage` with certain values from `MessageHeaders`. If available, values are mapped to the outbound mail's properties, such as the recipients (To, Cc, and

BCc), the from, the reply-to, and the subject. The header names are defined by the following constants:

```
MailHeaders.SUBJECT
MailHeaders.TO
MailHeaders.CC
MailHeaders.BCC
MailHeaders.FROM
MailHeaders.REPLY_TO
```

`MailHeaders` also lets you override corresponding `MailMessage` values. For example, if `MailMessage.to` is set to 'thing1@things.com' and the `MailHeaders.TO` message header is provided, it takes precedence and overrides the corresponding value in `MailMessage`.

25.2. Mail-receiving Channel Adapter

Spring Integration also provides support for inbound email with the `MailReceivingMessageSource`. It delegates to a configured instance of Spring Integration's own `MailReceiver` interface. There are two implementations: `Pop3MailReceiver` and `ImapMailReceiver`. The easiest way to instantiate either of these is by passing the 'uri' for a mail store to the receiver's constructor, as the following example shows:

```
MailReceiver receiver = new Pop3MailReceiver("pop3://usr:pwd@localhost/INBOX");
```

Another option for receiving mail is the IMAP `idle` command (if supported by your mail server). Spring Integration provides the `ImapIdleChannelAdapter`, which is itself a message-producing endpoint. It delegates to an instance of the `ImapMailReceiver` but enables asynchronous reception of mail messages. The next section has examples of configuring both types of inbound channel adapter with Spring Integration's namespace support in the 'mail' schema.

Normally, when the `IMAPMessage.getContent()` method is called, certain headers as well as the body are rendered (for a simple text email), as the following example shows:

```
To: thing1@things.com
From: thing2@morethings.com
Subject: Test Email

something
```

With a simple `MimeMessage`, `getContent()` returns the mail body (`something` in the preceding example).

Starting with version 2.2, the framework eagerly fetches IMAP messages and exposes them as an internal subclass of `MimeMessage`. This had the undesired side effect of changing the `getContent()` behavior. This inconsistency was further exacerbated by the Mail Mapping enhancement introduced in version 4.3, because, when a header mapper was provided, the payload was rendered by the `IMAPMessage.getContent()` method. This meant that the IMAP content differed, depending on whether or not a header mapper was provided.

Starting with version 5.0, messages originating from an IMAP source render the content in accordance with `IMAPMessage.getContent()` behavior, regardless of whether a header mapper is provided. If you do not use a header mapper and you wish to revert to the previous behavior of rendering only the body, set the `simpleContent` boolean property on the mail receiver to `true`. This property now controls the rendering regardless of whether a header mapper is used. It now allows body-only rendering when a header mapper is provided.

Starting with version 5.2, the `autoCloseFolder` option is provided on the mail receiver. Setting it to `false` doesn't close the folder automatically after a fetch, but instead an `IntegrationMessageHeaderAccessor.CLOSEABLE_RESOURCE` header (see `MessageHeaderAccessor` API for more information) is populated into every message to producer from the channel adapter. It is the target application's responsibility to call the `close()` on this header whenever it is necessary in the downstream flow:

```
Closeable closeableResource = StaticMessageHeaderAccessor.getCloseableResource
(mailMessage);
if (closeableResource != null) {
    closeableResource.close();
}
```

Keeping the folder open is useful in cases where communication with the server is needed during parsing multipart content of the email with attachments. The `close()` on the `IntegrationMessageHeaderAccessor.CLOSEABLE_RESOURCE` header delegates to the `AbstractMailReceiver` to close the folder with `expunge` option if `shouldDeleteMessages` is configured respectively on the `AbstractMailReceiver`.

25.3. Inbound Mail Message Mapping

By default, the payload of messages produced by the inbound adapters is the raw `MimeMessage`. You can use that object to interrogate the headers and content. Starting with version 4.3, you can provide a `HeaderMapper<MimeMessage>` to map the headers to `MessageHeaders`. For convenience, Spring Integration provides a `DefaultMailHeaderMapper` for this purpose. It maps the following headers:

- `mail_from`: A `String` representation of the `from` address.
- `mail_bcc`: A `String` array containing the `bcc` addresses.

- `mail_cc`: A `String` array containing the `cc` addresses.
- `mail_to`: A `String` array containing the `to` addresses.
- `mail_replyTo`: A `String` representation of the `replyTo` address.
- `mail_subject`: The mail subject.
- `mail_lineCount`: A line count (if available).
- `mail_receivedDate`: The received date (if available).
- `mail_size`: The mail size (if available).
- `mail_expunged`: A boolean indicating if the message is expunged.
- `mail_raw`: A `MultiValueMap` containing all the mail headers and their values.
- `mail_contentType`: The content type of the original mail message.
- `contentType`: The payload content type (see below).

When message mapping is enabled, the payload depends on the mail message and its implementation. Email contents are usually rendered by a `DataHandler` within the `MimeMessage`.

For a `text/*` email, the payload is a `String` and the `contentType` header is the same as `mail_contentType`.

For a messages with embedded `javax.mail.Part` instances, the `DataHandler` usually renders a `Part` object. These objects are not `Serializable` and are not suitable for serialization with alternative technologies such as `Kryo`. For this reason, by default, when mapping is enabled, such payloads are rendered as a raw `byte[]` containing the `Part` data. Examples of `Part` are `Message` and `Multipart`. The `contentType` header is `application/octet-stream` in this case. To change this behavior and receive a `Multipart` object payload, set `embeddedPartsAsBytes` to `false` on `MailReceiver`. For content types that are unknown to the `DataHandler`, the contents are rendered as a `byte[]` with a `contentType` header of `application/octet-stream`.

When you do not provide a header mapper, the message payload is the `MimeMessage` presented by `javax.mail`. The framework provides a `MailToStringTransformer` that you can use to convert the message by using a strategy to convert the mail contents to a `String`. This is also available by using the XML namespace, as the following example shows:

```
<int-mail:mail-to-string-transformer ... >
```

The following example does the same thing with Java configuration:

```
@Bean
@Transformer(inputChannel="...", outputChannel="...")
public Transformer transformer() {
    return new MailToStringTransformer();
}
```

The following example does the same thing with the Java DSL:

```
   ...
   .transform(Mail.toStringTransformer())
   ...
```

Starting with version 4.3, the transformer handles embedded `Part` instances (as well as `Multipart` instances, which were handled previously). The transformer is a subclass of `AbstractMailTransformer` that maps the address and subject headers from the preceding list. If you wish to perform some other transformation on the message, consider subclassing `AbstractMailTransformer`.

25.4. Mail Namespace Support

Spring Integration provides a namespace for mail-related configuration. To use it, configure the following schema locations:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int-mail="http://www.springframework.org/schema/integration/mail"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration/mail
    https://www.springframework.org/schema/integration/mail/spring-integration-
mail.xsd">
```

To configure an outbound channel adapter, provide the channel from which to receive and the MailSender, as the following example shows:

```
<int-mail:outbound-channel-adapter channel="outboundMail"
    mail-sender="mailSender"/>
```

Alternatively, you can provide the host, username, and password, as the following example shows:

```
<int-mail:outbound-channel-adapter channel="outboundMail"
    host="somehost" username="someuser" password="somepassword"/>
```

Starting with version 5.1.3, the `host`, `username` ane `mail-sender` can be omitted, if `java-mail-properties` is provided. However the `host` and `username` has to be configured with appropriate Java mail properties, e.g. for SMTP:

```
mail.user=someuser@gmail.com
mail.smtp.host=smtp.gmail.com
mail.smtp.port=587
```

As with any outbound Channel Adapter, if the referenced channel is a `PollableChannel`, you should provide a `<poller>` element (see Endpoint Namespace Support).

When you use the namespace support, you can also use a `header-enricher` message transformer. Doing so simplifies the application of the headers mentioned earlier to any message prior to sending to the mail outbound channel adapter.

The following example assumes the payload is a Java bean with appropriate getters for the specified properties, but you can use any SpEL expression:

```
<int-mail:header-enricher input-channel="expressionsInput" default-overwrite=
"false">
    <int-mail:to expression="payload.to"/>
    <int-mail:cc expression="payload.cc"/>
    <int-mail:bcc expression="payload.bcc"/>
    <int-mail:from expression="payload.from"/>
    <int-mail:reply-to expression="payload.replyTo"/>
    <int-mail:subject expression="payload.subject" overwrite="true"/>
</int-mail:header-enricher>
```

Alternatively, you can use the `value` attribute to specify a literal. You also can specify `default-overwrite` and individual `overwrite` attributes to control the behavior with existing headers.

To configure an inbound channel adapter, you have the choice between polling or event-driven (assuming your mail server supports IMAP `idle` — if not, then polling is the only option). A polling channel adapter requires the store URI and the channel to which to send inbound messages. The URI may begin with `pop3` or `imap`. The following example uses an `imap` URI:

```
<int-mail:inbound-channel-adapter id="imapAdapter"
      store-uri="imaps://[username]:[password]@imap.gmail.com/INBOX"
      java-mail-properties="javaMailProperties"
      channel="receiveChannel"
      should-delete-messages="true"
      should-mark-messages-as-read="true"
      auto-startup="true">
      <int:poller max-messages-per-poll="1" fixed-rate="5000"/>
</int-mail:inbound-channel-adapter>
```

If you do have IMAP `idle` support, you may want to configure the `imap-idle-channel-adapter` element instead. Since the `idle` command enables event-driven notifications, no poller is necessary for this adapter. It sends a message to the specified channel as soon as it receives the notification that new mail is available. The following example configures an IMAP `idle` mail channel:

```
<int-mail:imap-idle-channel-adapter id="customAdapter"
      store-uri="imaps://[username]:[password]@imap.gmail.com/INBOX"
      channel="receiveChannel"
      auto-startup="true"
      should-delete-messages="false"
      should-mark-messages-as-read="true"
      java-mail-properties="javaMailProperties"/>
```

You can provide `javaMailProperties` by creating and populating a regular `java.utils.Properties` object — for example, by using the `util` namespace provided by Spring.

If your username contains the '@' character, use '%40' instead of '@' to avoid parsing errors from the underlying JavaMail API.

The following example shows how to configure a `java.util.Properties` object:

```
<util:properties id="javaMailProperties">
  <prop key="mail.imap.socketFactory.class">javax.net.ssl.SSLSocketFactory</prop>
  <prop key="mail.imap.socketFactory.fallback">false</prop>
  <prop key="mail.store.protocol">imaps</prop>
  <prop key="mail.debug">false</prop>
</util:properties>
```

By default, the `ImapMailReceiver` searches for messages based on the default `SearchTerm`, which is all mail messages that:

- Are RECENT (if supported)

- Are NOT ANSWERED
- Are NOT DELETED
- Are NOT SEEN
- hHave not been processed by this mail receiver (enabled by the use of the custom USER flag or simply NOT FLAGGED if not supported)

The custom user flag is `spring-integration-mail-adapter`, but you can configure it. Since version 2.2, the `SearchTerm` used by the `ImapMailReceiver` is fully configurable with `SearchTermStrategy`, which you can inject by using the `search-term-strategy` attribute. A `SearchTermStrategy` is a strategy interface with a single method that lets you create an instance of the `SearchTerm` used by the `ImapMailReceiver`. The following listing shows the `SearchTermStrategy` interface:

```
public interface SearchTermStrategy {

    SearchTerm generateSearchTerm(Flags supportedFlags, Folder folder);

}
```

The following example relies `TestSearchTermStrategy` rather than the default `SearchTermStrategy`:

```
<mail:imap-idle-channel-adapter id="customAdapter"
            store-uri="imap:something"
            ...
            search-term-strategy="searchTermStrategy"/>

<bean id="searchTermStrategy"
  class=
"o.s.i.mail.config.ImapIdleChannelAdapterParserTests.TestSearchTermStrategy"/>
```

See Marking IMAP Messages When \Recent Is Not Supported for information about message flagging.

Important: IMAP PEEK

Starting with version 4.1.1, the IMAP mail receiver uses the `mail.imap.peek` or `mail.imaps.peek` JavaMail property, if specified. Previously, the receiver ignored the property and always set the `PEEK` flag. Now, if you explicitly set this property to `false`, the message ise marked as `\Seen` regardless of the setting of `shouldMarkMessagesRead`. If not specified, the previous behavior is retained (peek is `true`).

25.4.1. IMAP `idle` and Lost Connections

When using an IMAP `idle` channel adapter, connections to the server may be lost (for example, through network failure) and, since the JavaMail documentation explicitly states that the actual IMAP API is experimental, it is important to understand the differences in the API and how to deal with them when configuring IMAP `idle` adapters. Currently, Spring Integration mail adapters were tested with JavaMail 1.4.1 and JavaMail 1.4.3. Depending on which one is used, you must pay special attention to some of the JavaMail properties that need to be set with regard to auto-reconnect.

The following behavior was observed with Gmail but should provide you with some tips on how to solve re-connect issue with other providers. However feedback is always welcome. Again, the following notes are based on Gmail.

With JavaMail 1.4.1, if you set the `mail.imaps.timeout` property to a relatively short period of time (approximately 5 min in our testing), `IMAPFolder.idle()` throws `FolderClosedException` after this timeout. However, if this property is not set (it should be indefinite) the `IMAPFolder.idle()` method never returns and never throws an exception. It does, however, reconnect automatically if the connection was lost for a short period of time (under 10 min in our testing). However, if the connection was lost for a long period of time (over 10 min), `IMAPFolder.idle()`, does not throw `FolderClosedException` and does not re-establish the connection, and remains in the blocked state indefinitely, thus leaving you no possibility to reconnect without restarting the adapter. Consequently, the only way to make re-connecting work with JavaMail 1.4.1 is to set the `mail.imaps.timeout` property explicitly to some value, but it also means that such value should be relatively short (under 10 min) and the connection should be re-established relatively quickly. Again, it may be different with providers other than Gmail. With JavaMail 1.4.3 introduced significant improvements to the API, ensuring that there is always a condition that forces the `IMAPFolder.idle()` method to return `StoreClosedException` or `FolderClosedException` or to simply return, thus letting you proceed with auto-reconnecting. Currently auto-reconnecting runs infinitely making attempts to reconnect every ten seconds.

In both configurations, `channel` and `should-delete-messages` are required attributes. You should understand why `should-delete-messages` is required. The issue is with the POP3 protocol, which does not have any knowledge of messages that were read. It can only know what has been read within a single session. This means that, when your POP3 mail adapter runs, emails are successfully consumed as as they become available during each poll and no single email message is delivered more then once. However, as soon as you restart your adapter and begin a new session, all the email messages that might have been retrieved in the previous session are retrieved again. That is the nature of POP3. Some might argue that `should-delete-messages` should be `true` by default. In other words, there are two valid and mutually exclusive use that make it very hard to pick a single best default. You may want to configure your adapter as the only email receiver, in which case you want to be able to restart your adapter without fear that previously delivered messages are not delivered again. In this case, setting `should-delete-messages` to `true` would make the most sense. However, you may have another use case where you may want to have multiple adapters monitor email servers and their content. In other words, you want to 'peek but not touch'. Then setting `should-delete-messages` to `false` is much more appropriate. So since it is hard to choose what should be the right default value for the `should-delete-messages` attribute, we made it a required attribute to be set by you. Leaving it up to you also means that you are less likely to end up with unintended behavior.

When configuring a polling email adapter's `should-mark-messages-as-read` attribute, you should be aware of the protocol you are configuring to retrieve messages. For example, POP3 does not support this flag, which means setting it to either value has no effect, as messages are not marked as read.

In the case of a silently dropped connection, an idle cancel task is run in the background periodically (a new IDLE will usually immediately be processed). To control this interval, a `cancelIdleInterval` option is provided; default 120 (2 minutes). RFC 2177 recommends an interval no larger than 29 minutes.

You should understand that that these actions (marking messages read and deleting messages) are performed after the messages are received but before they are processed. This can cause messages to be lost.

You may wish to consider using transaction synchronization instead. See Transaction Synchronization.

The `<imap-idle-channel-adapter/>` also accepts the 'error-channel' attribute. If a downstream exception is thrown and an 'error-channel' is specified, a `MessagingException` message containing the failed message and the original exception is sent to this channel. Otherwise, if the downstream channels are synchronous, any such exception is logged as a warning by the channel adapter.

Beginning with the 3.0 release, the IMAP `idle` adapter emits application events (specifically `ImapIdleExceptionEvent` instances) when exceptions occur. This allows applications to detect and act on those exceptions. You can obtain the events by using an `<int-event:inbound-channel-adapter>` or any `ApplicationListener` configured to receive an `ImapIdleExceptionEvent` or one of its super classes.

25.5. Marking IMAP Messages When `\Recent` Is Not Supported

If `shouldMarkMessagesAsRead` is true, the IMAP adapters set the `\Seen` flag.

In addition, when an email server does not support the `\Recent` flag, the IMAP adapters mark messages with a user flag (by default, `spring-integration-mail-adapter`), as long as the server supports user flags. If not, `Flag.FLAGGED` is set to `true`. These flags are applied regardless of the `shouldMarkMessagesRead` setting.

As discussed in [search-term], the default `SearchTermStrategy` ignore messages that are so flagged.

Starting with version 4.2.2, you can set the name of the user flag by using `setUserFlag` on the `MailReceiver`. Doing so lets multiple receivers use a different flag (as long as the mail server supports user flags). The `user-flag` attribute is available when configuring the adapter with the namespace.

25.6. Email Message Filtering

Very often, you may encounter a requirement to filter incoming messages (for example, you want to read only emails that have 'Spring Integration' in the `Subject` line). You can accomplish this by connecting an inbound mail adapter with an expression-based `Filter`. Although it would work, there is a downside to this approach. Since messages would be filtered after going through the inbound mail adapter, all such messages would be marked as read (`SEEN`) or unread (depending on the value of `should-mark-messages-as-read` attribute). However, in reality, it be more useful to mark messages as `SEEN` only if they pass the filtering criteria. This is similar to looking at your email client while scrolling through all the messages in the preview pane, but only flagging messages that were actually opened and read as `SEEN`.

Spring Integration 2.0.4 introduced the `mail-filter-expression` attribute on `inbound-channel-adapter` and `imap-idle-channel-adapter`. This attribute lets you provide an expression that is a combination of SpEL and a regular expression. For example if you would like to read only emails that contain 'Spring Integration' in the subject line, you would configure the `mail-filter-expression` attribute like as follows: `mail-filter-expression="subject matches '(?i).Spring Integration."`.

Since `javax.mail.internet.MimeMessage` is the root context of the SpEL evaluation context, you can filter on any value available through `MimeMessage`, including the actual body of the message. This one is particularly important, since reading the body of the message typically results in such messages being marked as `SEEN` by default. However, since we now set the `PEEK` flag of every incoming message to 'true', only messages that were explicitly marked as `SEEN` are marked as read.

So, in the following example, only messages that match the filter expression are output by this adapter and only those messages are marked as read:

```
<int-mail:imap-idle-channel-adapter id="customAdapter"
    store-uri="imaps://some_google_address:${password}@imap.gmail.com/INBOX"
    channel="receiveChannel"
    should-mark-messages-as-read="true"
    java-mail-properties="javaMailProperties"
    mail-filter-expression="subject matches '(?i).*Spring Integration.*'"/>
```

In the preceding example, thanks to the `mail-filter-expression` attribute, only messages that contain 'Spring Integration' in the subject line are produced by this adapter.

Another reasonable question is what happens on the next poll or idle event or what happens when such an adapter is restarted. Can there be duplication of massages to be filtered? In other words, if, on the last retrieval where you had five new messages and only one passed the filter, what would happen with the other four? Would they go through the filtering logic again on the next poll or idle? After all, they were not marked as `SEEN`. The answer is no. They would not be subject to duplicate processing due to another flag (`RECENT`) that is set by the email server and is used by the Spring Integration mail search filter. Folder implementations set this flag to indicate that this message is new to this folder. That is, it has arrived since the last time this folder was opened. In other words, while our adapter may peek at the email, it also lets the email server know that such email was touched and should therefore be marked as `RECENT` by the email server.

25.7. Transaction Synchronization

Transaction synchronization for inbound adapters lets you take different actions after a transaction commits or rolls back. You can enable transaction synchronization by adding a `<transactional/>` element to the poller for the polled `<inbound-adapter/>` or to the `<imap-idle-inbound-adapter/>`. Even if there is no 'real' transaction involved, you can still enable this feature by using a `PseudoTransactionManager` with the `<transactional/>` element. For more information, see Transaction Synchronization.

Because of the many different mail servers and specifically the limitations that some have, at this time we provide only a strategy for these transaction synchronizations. You can send the messages to some other Spring Integration components or invoke a custom bean to perform some action. For example, to move an IMAP message to a different folder after the transaction commits, you might use something similar to the following:

```
<int-mail:imap-idle-channel-adapter id="customAdapter"
    store-uri="imaps://something.com:password@imap.something.com/INBOX"
    channel="receiveChannel"
    auto-startup="true"
    should-delete-messages="false"
    java-mail-properties="javaMailProperties">
    <int:transactional synchronization-factory="syncFactory"/>
</int-mail:imap-idle-channel-adapter>

<int:transaction-synchronization-factory id="syncFactory">
    <int:after-commit expression="@syncProcessor.process(payload)"/>
</int:transaction-synchronization-factory>

<bean id="syncProcessor" class="thing1.thing2.Mover"/>
```

The following example shows what the `Mover` class might look like:

```
public class Mover {

    public void process(MimeMessage message) throws Exception{
        Folder folder = message.getFolder();
        folder.open(Folder.READ_WRITE);
        String messageId = message.getMessageID();
        Message[] messages = folder.getMessages();
        FetchProfile contentsProfile = new FetchProfile();
        contentsProfile.add(FetchProfile.Item.ENVELOPE);
        contentsProfile.add(FetchProfile.Item.CONTENT_INFO);
        contentsProfile.add(FetchProfile.Item.FLAGS);
        folder.fetch(messages, contentsProfile);
        // find this message and mark for deletion
        for (int i = 0; i < messages.length; i++) {
            if (((MimeMessage) messages[i]).getMessageID().equals(messageId)) {
                messages[i].setFlag(Flags.Flag.DELETED, true);
                break;
            }
        }

        Folder somethingFolder = store.getFolder("SOMETHING"));
        somethingFolder.appendMessages(new MimeMessage[]{message});
        folder.expunge();
        folder.close(true);
        somethingFolder.close(false);
    }
}
```

For the message to be still available for manipulation after the transaction, *should-delete-messages* must be set to 'false'.

25.8. Configuring channel adapters with the Java DSL

To configure mail mail component in Java DSL, the framework provides a `o.s.i.mail.dsl.Mail` factory, which can be used like this:

```
@SpringBootApplication
public class MailApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(MailApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public IntegrationFlow imapMailFlow() {
        return IntegrationFlows
                .from(Mail.imapInboundAdapter("imap://user:pw@host:port/INBOX")
                            .searchTermStrategy(this::fromAndNotSeenTerm)
                            .userFlag("testSIUserFlag")
                            .simpleContent(true)
                            .javaMailProperties(p -> p.put("mail.debug", "false")
),
                    e -> e.autoStartup(true)
                            .poller(p -> p.fixedDelay(1000)))
                .channel(MessageChannels.queue("imapChannel"))
                .get();
    }

    @Bean
    public IntegrationFlow sendMailFlow() {
        return IntegrationFlows.from("sendMailChannel")
                .enrichHeaders(Mail.headers()
                        .subjectFunction(m -> "foo")
                        .from("foo@bar")
                        .toFunction(m -> new String[] { "bar@baz" }))
                .handle(Mail.outboundAdapter("gmail")
                            .port(smtpServer.getPort())
                            .credentials("user", "pw")
                            .protocol("smtp")),
                    e -> e.id("sendMailEndpoint"))
                .get();
    }
}
```

Chapter 26. MongoDb Support

Version 2.1 introduced support for MongoDB: a "high-performance, open source, document-oriented database".

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-mongodb</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-mongodb:5.3.0.RC1"
```

To download, install, and run MongoDB, see the MongoDB documentation.

26.1. Connecting to MongoDb

26.1.1. Blocking or Reactive?

Beginning with version 5.3, Spring Integration provides support for reactive MongoDB drivers to enable non-blocking I/O when accessing MongoDB. To enable reactive support, add the MongoDB reactive streams driver to your dependencies:

Maven

```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-reactivestreams</artifactId>
  </dependency>
```

Gradle

```
compile "org.mongodb:mongodb-driver-reactivestreams"
```

For regular synchronous client you need to add its respective driver into dependencies:

Maven

```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-sync</artifactId>
  </dependency>
```

Gradle

```
compile "org.mongodb:mongodb-driver-sync"
```

Both of them are `optional` in the framework for better end-user choice support.

To begin interacting with MongoDB, you first need to connect to it. Spring Integration builds on the support provided by another Spring project, Spring Data MongoDB. It provides factory classes called `MongoDatabaseFactory` and `ReactiveMongoDatabaseFactory`, which simplify integration with the MongoDB Client API.

Spring Data provides provides the blocking MongoDB driver by default but you may opt-in for reactive usage by including the above dependency.

26.1.2. Using `MongoDatabaseFactory`

To connect to MongoDB you can use an implementation of the `MongoDatabaseFactory` interface.

The following example shows how to use `SimpleMongoClientDatabaseFactory`, the out-of-the-box implementation, in Java:

```
MongoDatabaseFactory mongoDbFactory =
        new SimpleMongoClientDatabaseFactory(com.mongodb.client.MongoClients
.create(), "test");
```

The following example shows how to use `SimpleMongoClientDatabaseFactory` in XML configuration:

```
<bean id="mongoDbFactory" class=
"o.s.data.mongodb.core.SimpleMongoClientDatabaseFactory">
    <constructor-arg>
        <bean class="com.mongodb.client.MongoClients" factory-method="create"/>
    </constructor-arg>
    <constructor-arg value="test"/>
</bean>
```

`SimpleMongoClientDatabaseFactory` takes two arguments: a `MongoClient` instance and a `String` that

specifies the name of the database. If you need to configure properties such as `host`, `port`, and others, you can pass those by using one of the constructors provided by the underlying `MongoClients` class. For more information on how to configure MongoDB, see the Spring-Data-MongoDB reference.

26.1.3. Using `ReactiveMongoDatabaseFactory`

To connect to MongoDB with the reactive driver, you can use an implementation of the `ReactiveMongoDatabaseFactory` interface.

The following example shows how to use `SimpleReactiveMongoDatabaseFactory`, the out-of-the-box implementation, in Java:

```
new SimpleReactiveMongoDatabaseFactory(com.mongodb.reactivestreams.client
.MongoClients.create(), "test");
```

The following example shows how to use `SimpleReactiveMongoDatabaseFactory` in XML configuration:

```
<bean id="mongoDbFactory" class=
"o.s.data.mongodb.core.SimpleReactiveMongoDatabaseFactory">
    <constructor-arg>
        <bean class="com.mongodb.reactivestreams.client.MongoClients" factory-
method="create"/>
    </constructor-arg>
    <constructor-arg value="test"/>
</bean>
```

26.2. MongoDB Message Store

As described in the *Enterprise Integration Patterns* (EIP) book, a Message Store lets you persist messages. Doing so can be useful when dealing with components that have the ability to buffer messages (`QueueChannel`, `aggregator`, `resequencer`, and others.) if reliability is a concern. In Spring Integration, the `MessageStore` strategy also provides the foundation for the claim check pattern, which is described in EIP as well.

Spring Integration's MongoDB module provides the `MongoDbMessageStore`, which is an implementation of both the `MessageStore` strategy (mainly used by the claim check pattern) and the `MessageGroupStore` strategy (mainly used by the aggregator and resequencer patterns).

The following example configures a `MongoDbMessageStore` to use a `QueueChannel` and an `aggregator`:

```
<bean id="mongoDbMessageStore" class="o.s.i.mongodb.store.MongoDbMessageStore">
    <constructor-arg ref="mongoDbFactory"/>
</bean>

<int:channel id="somePersistentQueueChannel">
    <int:queue message-store="mongoDbMessageStore"/>
<int:channel>

<int:aggregator input-channel="inputChannel" output-channel="outputChannel"
         message-store="mongoDbMessageStore"/>
```

The preceding example is a simple bean configuration, and it expects a `MongoDbFactory` as a constructor argument.

The `MongoDbMessageStore` expands the `Message` as a Mongo document with all nested properties by using the Spring Data Mongo mapping mechanism. It is useful when you need to have access to the `payload` or `headers` for auditing or analytics — for example, against stored messages.

The `MongoDbMessageStore` uses a custom `MappingMongoConverter` implementation to store `Message` instances as MongoDB documents, and there are some limitations for the properties (`payload` and `header` values) of the `Message`.

Starting with version 5.1.6, the `MongoDbMessageStore` can be configured with custom converters which are propagated into an internal `MappingMongoConverter` implementation. See `MongoDbMessageStore.setCustomConverters(Object… customConverters)` JavaDocs for more information.

Spring Integration 3.0 introduced the `ConfigurableMongoDbMessageStore`. It implements both the `MessageStore` and `MessageGroupStore` interfaces. This class can receive, as a constructor argument, a `MongoTemplate`, with which you can, for example, configure a custom `WriteConcern`. Another constructor requires a `MappingMongoConverter` and a `MongoDbFactory`, which lets you provide some custom conversions for `Message` instances and their properties. Note that, by default, the `ConfigurableMongoDbMessageStore` uses standard Java serialization to write and read `Message` instances to and from MongoDB (see `MongoDbMessageBytesConverter`) and relies on default values for other properties from `MongoTemplate`. It builds a `MongoTemplate` from the provided `MongoDbFactory` and `MappingMongoConverter`. The default name for the collection stored by the `ConfigurableMongoDbMessageStore` is `configurableStoreMessages`. We recommend using this implementation to create robust and flexible solutions when messages contain complex data types.

26.2.1. MongoDB Channel Message Store

Version 4.0 introduced the new `MongoDbChannelMessageStore`. It is an optimized `MessageGroupStore` for use in `QueueChannel` instances. With `priorityEnabled = true`, you can use it in `<int:priority-queue>` instances to achieve priority-order polling for persisted messages. The priority MongoDB document field is populated from the `IntegrationMessageHeaderAccessor.PRIORITY` (`priority`) message header.

In addition, all MongoDB `MessageStore` instances now have a `sequence` field for `MessageGroup` documents. The `sequence` value is the result of an `$inc` operation for a simple `sequence` document from the same collection, which is created on demand. The `sequence` field is used in `poll` operations to provide first-in-first-out (FIFO) message order (within priority, if configured) when messages are stored within the same millisecond.

We do not recommend using the same `MongoDbChannelMessageStore` bean for priority and non-priority, because the `priorityEnabled` option applies to the entire store. However, the same `collection` can be used for both `MongoDbChannelMessageStore` types, because message polling from the store is sorted and uses indexes. To configure that scenario, you can extend one message store bean from the other, as the following example shows:

```
<bean id="channelStore" class="o.s.i.mongodb.store.MongoDbChannelMessageStore">
    <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
</bean>

<int:channel id="queueChannel">
    <int:queue message-store="store"/>
</int:channel>

<bean id="priorityStore" parent="channelStore">
    <property name="priorityEnabled" value="true"/>
</bean>

<int:channel id="priorityChannel">
    <int:priority-queue message-store="priorityStore"/>
</int:channel>
```

26.2.2. MongoDB Metadata Store

Spring Integration 4.2 introduced a new MongoDB-based `MetadataStore` (see Metadata Store) implementation. You can use the `MongoDbMetadataStore` to maintain metadata state across application restarts. You can use this new `MetadataStore` implementation with adapters such as:

- Feed
- File
- FTP
- SFTP

To instruct these adapters to use the new `MongoDbMetadataStore`, declare a Spring bean with a bean name of `metadataStore`. The feed inbound channel adapter automatically picks up and use the declared `MongoDbMetadataStore`. The following example shows how to declare a bean with a name of `metadataStore`:

```
@Bean
public MetadataStore metadataStore(MongoDbFactory factory) {
    return new MongoDbMetadataStore(factory, "integrationMetadataStore");
}
```

The `MongoDbMetadataStore` also implements `ConcurrentMetadataStore`, letting it be reliably shared across multiple application instances, where only one instance is allowed to store or modify a key's value. All these operations are atomic, thanks to MongoDB guarantees.

26.3. MongoDB Inbound Channel Adapter

The MongoDB inbound channel adapter is a polling consumer that reads data from MongoDB and sends it as a `Message` payload. The following example shows how to configure a MongoDB inbound channel adapter:

```
<int-mongodb:inbound-channel-adapter id="mongoInboundAdapter"
       channel="replyChannel"
       query="{'name' : 'Bob'}"
       entity-class="java.lang.Object"
       auto-startup="false">
        <int:poller fixed-rate="100"/>
</int-mongodb:inbound-channel-adapter>
```

As the preceding configuration shows, you configure a MongoDb inbound channel adapter by using the `inbound-channel-adapter` element and providing values for various attributes, such as:

- `query`: A JSON query (see MongoDB Querying)
- `query-expression`: A SpEL expression that is evaluated to a JSON query string (as the `query` attribute above) or to an instance of `o.s.data.mongodb.core.query.Query`. Mutually exclusive with the `query` attribute.
- `entity-class`: The type of the payload object. If not supplied, a `com.mongodb.DBObject` is returned.
- `collection-name` or `collection-name-expression`: Identifies the name of the MongoDB collection to use.
- `mongodb-factory`: Reference to an instance of `o.s.data.mongodb.MongoDbFactory`
- `mongo-template`: Reference to an instance of `o.s.data.mongodb.core.MongoTemplate`
- Other attributes that are common across all other inbound adapters (such as 'channel').

You cannot set both `mongo-template` and `mongodb-factory`.

The preceding example is relatively simple and static, since it has a literal value for the `query` and uses the default name for a `collection`. Sometimes, you may need to change those values at

runtime, based on some condition. To do so, use their `-expression` equivalents (`query-expression` and `collection-name-expression`), where the provided expression can be any valid SpEL expression.

Also, you may wish to do some post-processing to the successfully processed data that was read from the MongoDB. For example; you may want to move or remove a document after it has been processed. You can do so by using that transaction synchronization feature Spring Integration 2.2 added, as the following example shows:

```
<int-mongodb:inbound-channel-adapter id="mongoInboundAdapter"
    channel="replyChannel"
    query-expression="new BasicQuery('{''name'' : ''Bob''}').limit(100)"
    entity-class="java.lang.Object"
    auto-startup="false">
        <int:poller fixed-rate="200" max-messages-per-poll="1">
            <int:transactional synchronization-factory="syncFactory"/>
        </int:poller>
</int-mongodb:inbound-channel-adapter>

<int:transaction-synchronization-factory id="syncFactory">
    <int:after-commit
        expression="@documentCleaner.remove(#mongoTemplate, payload,
headers.mongo_collectionName)"
        channe="someChannel"/>
</int:transaction-synchronization-factory>

<bean id="documentCleaner" class="thing1.thing2.DocumentCleaner"/>

<bean id="transactionManager" class="o.s.i.transaction.PseudoTransactionManager"/>
```

The following example shows the `DocumentCleaner` referenced in the preceding example:

```
public class DocumentCleaner {
    public void remove(MongoOperations mongoOperations, Object target, String
collectionName) {
        if (target instanceof List<?>){
            List<?> documents = (List<?>) target;
            for (Object document : documents) {
                mongoOperations.remove(new BasicQuery(JSON.serialize(document)),
collectionName);
            }
        }
    }
}
```

You can declare your poller to be transactional by using the `transactional` element. This element

can reference a real transaction manager (for example, if some other part of your flow invokes JDBC). If you do not have a "real" transaction, you can use an instance of `o.s.i.transaction.PseudoTransactionManager`, which is an implementation of Spring's `PlatformTransactionManager` and enables the use of the transaction synchronization features of the Mongo adapter when there is no actual transaction.

Doing so does not make MongoDB itself transactional. It lets the synchronization of actions be taken before or after success (commit) or after failure (rollback).

Once your poller is transactional, you can set an instance of the `o.s.i.transaction.TransactionSynchronizationFactory` on the `transactional` element. A `TransactionSynchronizationFactory` creates an instance of the `TransactionSynchronization`. For your convenience, we have exposed a default SpEL-based `TransactionSynchronizationFactory` that lets you configure SpEL expressions, with their execution being coordinated (synchronized) with a transaction. Expressions for before-commit, after-commit, and after-rollback events are supported, together with a channel for each event where the evaluation result (if any) is sent. For each child element, you can specify `expression` and `channel` attributes. If only the `channel` attribute is present, the received message is sent there as part of the particular synchronization scenario. If only the `expression` attribute is present and the result of an expression is a non-null value, a message with the result as the payload is generated and sent to a default channel (`NullChannel`) and appears in the logs (on the `DEBUG` level). If you want the evaluation result to go to a specific channel, add a `channel` attribute. If the result of an expression is null or void, no message is generated.

For more information about transaction synchronization, see Transaction Synchronization.

26.4. MongoDB Change Stream Inbound Channel Adapter

Starting with version 5.3, the `spring-integration-mongodb` module introduces the `MongoDbChangeStreamMessageProducer` - a reactive `MessageProducerSupport` implementation for the Spring Data `ReactiveMongoOperations.changeStream(String, ChangeStreamOptions, Class)` API. This component produces a `Flux` of messages with a `body` of `ChangeStreamEvent` as the payload by default and some change stream related headers (see `MongoHeaders`). It is recommended that this `MongoDbChangeStreamMessageProducer` is combined with a `FluxMessageChannel` as the `outputChannel` for on-demand subscription and event consumption downstream.

The Java DSL configuration for this channel adapter may look like this:

```
@Bean
IntegrationFlow changeStreamFlow(ReactiveMongoOperations mongoTemplate) {
    return IntegrationFlows.from(
            MongoDb.changeStreamInboundChannelAdapter(mongoTemplate)
                    .domainType(Person.class)
                    .collection("person")
                    .extractBody(false))
            .channel(MessageChannels.flux())
            .get();
}
```

When the `MongoDbChangeStreamMessageProducer` is stopped, or the subscription is cancelled downstream, or the MongoDb change stream produces an `OperationType.INVALIDATE`, the `Publisher` is completed. The channel adapter can be started again and a new `Publisher` of source data is created and it is automatically subscribed in the `MessageProducerSupport.subscribeToPublisher(Publisher<? extends Message<?>>)`. This channel adapter can be reconfigured for new options between starts, if there is a requirement to consume change stream events from other places.

See more information about change stream support in Spring Data MongoDb documentation.

26.5. MongoDB Outbound Channel Adapter

The MongoDB outbound channel adapter lets you write the message payload to a MongoDB document store, as the following example shows:

```
<int-mongodb:outbound-channel-adapter id="fullConfigWithCollectionExpression"
    collection-name="myCollection"
    mongo-converter="mongoConverter"
    mongodb-factory="mongoDbFactory" />
```

As the preceding configuration shows, you can configure a MongoDB outbound channel adapter by using the `outbound-channel-adapter` element, providing values for various attributes, such as:

- `collection-name` or `collection-name-expression`: Identifies the name of the MongoDb collection to use.
- `mongo-converter`: Reference to an instance of `o.s.data.mongodb.core.convert.MongoConverter` that assists with converting a raw Java object to a JSON document representation.
- `mongodb-factory`: Reference to an instance of `o.s.data.mongodb.MongoDbFactory`.
- `mongo-template`: Reference to an instance of `o.s.data.mongodb.core.MongoTemplate`. NOTE: you can not have both mongo-template and mongodb-factory set.
- Other attributes that are common across all other inbound adapters (such as 'channel').

The preceding example is relatively simple and static, since it has a literal value for the `collection-name`. Sometimes, you may need to change this value at runtime, based on some condition. To do that, use `collection-name-expression`, where the provided expression is any valid SpEL expression.

26.6. MongoDB Outbound Gateway

Version 5.0 introduced the MongoDB outbound gateway. It allows you query a database by sending a message to its request channel. The gateway then send the response to the reply channel. You can use the message payload and headers to specify the query and the collection name, as the following example shows:

```
<int-mongodb:outbound-gateway id="gatewayQuery"
    mongodb-factory="mongoDbFactory"
    mongo-converter="mongoConverter"
    query="{firstName: 'Bob'}"
    collection-name="myCollection"
    request-channel="in"
    reply-channel="out"
    entity-class="org.springframework.integration.mongodb.test.entity$Person"/>
```

You can use the following attributes with a MongoDB outbound Gateway:

- `collection-name` or `collection-name-expression`: Identifies the name of the MongoDB collection to use.
- `mongo-converter`: Reference to an instance of `o.s.data.mongodb.core.convert.MongoConverter` that assists with converting a raw Java object to a JSON document representation.
- `mongodb-factory`: Reference to an instance of `o.s.data.mongodb.MongoDbFactory`.
- `mongo-template`: Reference to an instance of `o.s.data.mongodb.core.MongoTemplate`. NOTE: you can not set both `mongo-template` and `mongodb-factory`.
- `entity-class`: The fully qualified name of the entity class to be passed to the `find(..)` and `findOne(..)` methods in MongoTemplate. If this attribute is not provided, the default value is `org.bson.Document`.
- `query` or `query-expression`: Specifies the MongoDB query. See the MongoDB documentation for more query samples.
- `collection-callback`: Reference to an instance of `org.springframework.data.mongodb.core.CollectionCallback`. Preferable an instance of `o.s.i.mongodb.outbound.MessageCollectionCallback` since 5.0.11 with the request message context. See its Javadocs for more information. NOTE: You can not have both `collection-callback` and any of the query attributes.

26.6.1. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the outbound gateway

with Java configuration:

```
@SpringBootApplication
public class MongoDbJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(MongoDbJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Autowired
    private MongoDbFactory mongoDbFactory;

    @Bean
    @ServiceActivator(inputChannel = "requestChannel")
    public MessageHandler mongoDbOutboundGateway() {
        MongoDbOutboundGateway gateway = new MongoDbOutboundGateway(this
.mongoDbFactory);
        gateway.setCollectionNameExpressionString("'myCollection'");
        gateway.setQueryExpressionString("'{''name'' : ''Bob''}'");
        gateway.setExpectSingleResult(true);
        gateway.setEntityClass(Person.class);
        gateway.setOutputChannelName("replyChannel");
        return gateway;
    }

    @Bean
    @ServiceActivator(inputChannel = "replyChannel")
    public MessageHandler handler() {
        return message -> System.out.println(message.getPayload());
    }
}
```

26.6.2. Configuring with the Java DSL

The following Spring Boot application show an example of how to configure the outbound gateway with the Java DSL:

```
@SpringBootApplication
public class MongoDbJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(MongoDbJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Autowired
    private MongoDbFactory;

    @Autowired
    private MongoConverter;


    @Bean
    public IntegrationFlow gatewaySingleQueryFlow() {
        return f -> f
                .handle(queryOutboundGateway())
                .channel(c -> c.queue("retrieveResults"));
    }

    private MongoDbOutboundGatewaySpec queryOutboundGateway() {
        return MongoDb.outboundGateway(this.mongoDbFactory, this.mongoConverter)
                .query("{name : 'Bob'}")
                .collectionNameFunction(m -> m.getHeaders().get("collection"))
                .expectSingleResult(true)
                .entityClass(Person.class);
    }

}
```

As an alternate to the `query` and `query-expression` properties, you can specify other database operations by using the `collectionCallback` property as a reference to the `MessageCollectionCallback` functional interface implementation. The following example specifies a count operation:

```
private MongoDbOutboundGatewaySpec collectionCallbackOutboundGateway() {
    return MongoDb.outboundGateway(this.mongoDbFactory, this.mongoConverter)
            .collectionCallback((collection, requestMessage) -> collection.count(
))
            .collectionName("myCollection");
    }
```

26.7. MongoDB Reactive Channel Adapters

Starting with version 5.3, the `ReactiveMongoDbStoringMessageHandler` and `ReactiveMongoDbMessageSource` implementations are provided. They are based on the `ReactiveMongoOperations` from Spring Data and requires a `org.mongodb:mongodb-driver-reactivestreams` dependency.

The `ReactiveMongoDbStoringMessageHandler` is an implementation of the `ReactiveMessageHandler` which is supported natively in the framework when reactive streams composition is involved in the integration flow definition. See more information in the ReactiveMessageHandler.

From configuration perspective there is no difference with many other standard channel adapters. For example with Java DSL such a channel adapter could be used like:

```
@Bean
public IntegrationFlow reactiveMongoDbFlow(ReactiveMongoDatabaseFactory
mongoDbFactory) {
    return f -> f
            .channel(MessageChannels.flux())
            .handle(MongoDb.reactiveOutboundChannelAdapter(mongoDbFactory));
}
```

In this sample we are going to connect to the MongoDb via provided `ReactiveMongoDatabaseFactory` and store a data from request message into a default collection with the `data` name. The real operation is going to be performed on-demand from the reactive stream composition in the internally created `ReactiveStreamsConsumer`.

The `ReactiveMongoDbMessageSource` is an `AbstractMessageSource` implementation based on the provided `ReactiveMongoDatabaseFactory` or `ReactiveMongoOperations` and MongoDb query (or expression), calls `find()` or `findOne()` operation according an `expectSingleResult` option with an expected `entityClass` type to convert a query result. A query execution and result evaluation is performed on demand when `Publisher` (`Flux` or `Mono` according `expectSingleResult` option) in the payload of produced message is subscribed. The framework can subscribe to such a payload automatically (essentially `flatMap`) when splitter and `FluxMessageChannel` are used downstream. Otherwise it is target application responsibility to subscribe into a polled publishers in downstream endpoints.

With Java DSL such a channel adapter could be configured like:

```
@Bean
public IntegrationFlow reactiveMongoDbFlow(ReactiveMongoDatabaseFactory
mongoDbFactory) {
    return IntegrationFlows
            .from(MongoDb.reactiveInboundChannelAdapter(mongoDbFactory, "{'name' :
'Name'}")
                            .entityClass(Person.class),
                    c -> c.poller(Pollers.fixedDelay(1000)))
            .split()
            .channel(c -> c.flux("output"))
            .get();
}
```

Chapter 27. MQTT Support

Spring Integration provides inbound and outbound channel adapters to support the Message Queueing Telemetry Transport (MQTT) protocol.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-mqtt</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-mqtt:5.3.0.RC1"
```

The current implementation uses the Eclipse Paho MQTT Client library.

Configuration of both adapters is achieved using the `DefaultMqttPahoClientFactory`. Refer to the Paho documentation for more information about configuration options.

We recommend configuring an `MqttConnectOptions` object and injecting it into the factory, instead of setting the (deprecated) options on the factory itself.

27.1. Inbound (Message-driven) Channel Adapter

The inbound channel adapter is implemented by the `MqttPahoMessageDrivenChannelAdapter`. For convenience, you can configure it by using the namespace. A minimal configuration might be as follows:

```
<bean id="clientFactory"
        class=
"org.springframework.integration.mqtt.core.DefaultMqttPahoClientFactory">
    <property name="connectionOptions">
        <bean class="org.eclipse.paho.client.mqttv3.MqttConnectOptions">
            <property name="userName" value="${mqtt.username}"/>
            <property name="password" value="${mqtt.password}"/>
        </bean>
    </property>
</bean>

<int-mqtt:message-driven-channel-adapter id="mqttInbound"
    client-id="${mqtt.default.client.id}.src"
    url="${mqtt.url}"
    topics="sometopic"
    client-factory="clientFactory"
    channel="output"/>
```

The following listing shows the available attributes:

```
<int-mqtt:message-driven-channel-adapter id="oneTopicAdapter"
    client-id="foo"  ①
    url="tcp://localhost:1883"  ②
    topics="bar,baz"  ③
    qos="1,2"  ④
    converter="myConverter"  ⑤
    client-factory="clientFactory"  ⑥
    send-timeout="123"  ⑦
    error-channel="errors"  ⑧
    recovery-interval="10000"  ⑨
    manual-acks="false" ⑩
    channel="out" />
```

① The client ID.

② The broker URL.

③ A comma-separated list of topics from which this adapter receives messages.

④ A comma-separated list of QoS values. It can be a single value that is applied to all topics or a value for each topic (in which case, the lists must be the same length).

⑤ An `MqttMessageConverter` (optional). By default, the default `DefaultPahoMessageConverter` produces a message with a `String` payload with the following headers:

- `mqtt_topic`: The topic from which the message was received
- `mqtt_duplicate`: `true` if the message is a duplicate
- `mqtt_qos`: The quality of service You can configure the `DefaultPahoMessageConverter` to return the raw `byte[]` in the payload by declaring it as a `<bean/>` and setting the `payloadAsBytes` property to `true`.

⑥ The client factory.

⑦ The send timeout. It applies only if the channel might block (such as a bounded `QueueChannel` that is currently full).

⑧ The error channel. Downstream exceptions are sent to this channel, if supplied, in an `ErrorMessage`. The payload is a `MessagingException` that contains the failed message and cause.

⑨ The recovery interval. It controls the interval at which the adapter attempts to reconnect after a failure. It defaults to `10000ms` (ten seconds).

⑩ The acknowledgment mode; set to true for manual acknowledgment.

Starting with version 4.1, you can omit the URL. Instead, you can provide the server URIs in the `serverURIs` property of the `DefaultMqttPahoClientFactory`. Doing so enables, for example, connection to a highly available (HA) cluster.

Starting with version 4.2.2, an `MqttSubscribedEvent` is published when the adapter successfully subscribes to the topics. `MqttConnectionFailedEvent` events are published when the connection or

subscription fails. These events can be received by a bean that implements `ApplicationListener`.

Also, a new property called `recoveryInterval` controls the interval at which the adapter attempts to reconnect after a failure. It defaults to `10000ms` (ten seconds).

Prior to version 4.2.3, the client always unsubscribed when the adapter was stopped. This was incorrect because, if the client QOS is greater than 0, we need to keep the subscription active so that messages arriving while the adapter is stopped are delivered on the next start. This also requires setting the `cleanSession` property on the client factory to `false`. It defaults to `true`.

Starting with version 4.2.3, the adapter does not unsubscribe (by default) if the `cleanSession` property is `false`.

This behavior can be overridden by setting the `consumerCloseAction` property on the factory. It can have values: `UNSUBSCRIBE_ALWAYS`, `UNSUBSCRIBE_NEVER`, and `UNSUBSCRIBE_CLEAN`. The latter (the default) unsubscribes only if the `cleanSession` property is `true`.

To revert to the pre-4.2.3 behavior, use `UNSUBSCRIBE_ALWAYS`.

Starting with version 5.0, the `topic`, `qos`, and `retained` properties are mapped to `.RECEIVED_…` headers (`MqttHeaders.RECEIVED_TOPIC`, `MqttHeaders.RECEIVED_QOS`, and `MqttHeaders.RECEIVED_RETAINED`), to avoid inadvertent propagation to an outbound message that (by default) uses the `MqttHeaders.TOPIC`, `MqttHeaders.QOS`, and `MqttHeaders.RETAINED` headers.

27.1.1. Adding and Removing Topics at Runtime

Starting with version 4.1, you can programmatically change the topics to which the adapter is subscribed. Spring Integration provides the `addTopic()` and `removeTopic()` methods. When adding topics, you can optionally specify the `QoS` (default: 1). You can also modify the topics by sending an appropriate message to a `<control-bus/>` with an appropriate payload — for example: `"myMqttAdapter.addTopic('foo', 1)"`.

Stopping and starting the adapter has no effect on the topic list (it does not revert to the original settings in the configuration). The changes are not retained beyond the life cycle of the application context. A new application context reverts to the configured settings.

Changing the topics while the adapter is stopped (or disconnected from the broker) takes effect the next time a connection is established.

27.1.2. Manual Acks

Starting with version 5.3, you can set the `manualAcks` property to true. Often used to asynchronously acknowledge delivery. When set to `true`, header (`IntegrationMessageHeaderAccessor.ACKNOWLEDGMENT_CALLBACK`) is added to the message with the value being a `SimpleAcknowledgment`. You must invoke the `acknowledge()` method to complete the delivery. See the Javadocs for `IMqppClient setManualAcks()` and `messageArrivedComplete()` for more

information. For convenience a header accessor is provided:

```
StaticMessageHeaderAccessor.acknowledgment(someMessage).acknowledge();
```

27.1.3. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the inbound adapter with Java configuration:

```
@SpringBootApplication
public class MqttJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(MqttJavaApplication.class)
                .web(false)
                .run(args);
    }

    @Bean
    public MessageChannel mqttInputChannel() {
        return new DirectChannel();
    }

    @Bean
    public MessageProducer inbound() {
        MqttPahoMessageDrivenChannelAdapter adapter =
                new MqttPahoMessageDrivenChannelAdapter("tcp://localhost:1883",
"testClient",
                                                        "topic1", "topic2");
        adapter.setCompletionTimeout(5000);
        adapter.setConverter(new DefaultPahoMessageConverter());
        adapter.setQos(1);
        adapter.setOutputChannel(mqttInputChannel());
        return adapter;
    }

    @Bean
    @ServiceActivator(inputChannel = "mqttInputChannel")
    public MessageHandler handler() {
        return new MessageHandler() {

            @Override
            public void handleMessage(Message<?> message) throws
MessagingException {
                System.out.println(message.getPayload());
            }

        };
    }

}
```

27.1.4. Configuring with the Java DSL

The following Spring Boot application provides an example of configuring the inbound adapter with the Java DSL:

```
@SpringBootApplication
public class MqttJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(MqttJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public IntegrationFlow mqttInbound() {
        return IntegrationFlows.from(
                        new MqttPahoMessageDrivenChannelAdapter(
"tcp://localhost:1883",
                                            "testClient", "topic1", "topic2");)
                .handle(m -> System.out.println(m.getPayload()))
                .get();
    }

}
```

27.2. Outbound Channel Adapter

The outbound channel adapter is implemented by the `MqttPahoMessageHandler`, which is wrapped in a `ConsumerEndpoint`. For convenience, you can configure it by using the namespace.

Starting with version 4.1, the adapter supports asynchronous send operations, avoiding blocking until the delivery is confirmed. You can emit application events to enable applications to confirm delivery if desired.

The following listing shows the attributes available for an outbound channel adapter:

```
<int-mqtt:outbound-channel-adapter id="withConverter"
    client-id="foo"  ①
    url="tcp://localhost:1883"  ②
    converter="myConverter"  ③
    client-factory="clientFactory"  ④
    default-qos="1"  ⑤
    qos-expression="" ⑥
    default-retained="true"  ⑦
    retained-expression="" ⑧
    default-topic="bar"  ⑨
    topic-expression="" ⑩
    async="false"  ⑪
    async-events="false"  ⑫
    channel="target" />
```

① The client ID.

② The broker URL.

③ An `MqttMessageConverter` (optional). The default `DefaultPahoMessageConverter` recognizes the following headers:

- `mqtt_topic`: The topic to which the message will be sent
- `mqtt_retained`: `true` if the message is to be retained
- `mqtt_qos`: The quality of service

④ The client factory.

⑤ The default quality of service. It is used if no `mqtt_qos` header is found or the `qos-expression` returns `null`. It is not used if you supply a custom `converter`.

⑥ An expression to evaluate to determine the qos. The default is `headers[mqtt_qos]`.

⑦ The default value of the retained flag. It is used if no `mqtt_retained` header is found. It is not used if a custom `converter` is supplied.

⑧ An expression to evaluate to determine the retained boolean. The default is `headers[mqtt_retained]`.

⑨ The default topic to which the message is sent (used if no `mqtt_topic` header is found).

⑩ An expression to evaluate to determine the destination topic. The default is `headers['mqtt_topic']`.

⑪ When `true`, the caller does not block. Rather, it waits for delivery confirmation when a message is sent. The default is `false` (the send blocks until delivery is confirmed).

⑫ When `async` and `async-events` are both `true`, an `MqttMessageSentEvent` is emitted (See Events). It contains the message, the topic, the `messageId` generated by the client library, the `clientId`, and the `clientInstance` (incremented each time the client is connected). When the delivery is confirmed by the client library, an `MqttMessageDeliveredEvent` is emitted. It contains the the `messageId`, the `clientId`, and the `clientInstance`, enabling delivery to be correlated with the send. Any `ApplicationListener` or an event inbound channel adapter can received these

events. Note that it is possible for the `MqttMessageDeliveredEvent` to be received before the `MqttMessageSentEvent`. The default is `false`.

Starting with version 4.1, the URL can be omitted. Instead, the server URIs can be provided in the `serverURIs` property of the `DefaultMqttPahoClientFactory`. This enables, for example, connection to a highly available (HA) cluster.

27.2.1. Configuring with Java Configuration

The following Spring Boot application show an example of how to configure the outbound adapter with Java configuration:

```
@SpringBootApplication
@IntegrationComponentScan
public class MqttJavaApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
                new SpringApplicationBuilder(MqttJavaApplication.class)
                        .web(false)
                        .run(args);
        MyGateway gateway = context.getBean(MyGateway.class);
        gateway.sendToMqtt("foo");
    }

    @Bean
    public MqttPahoClientFactory mqttClientFactory() {
        DefaultMqttPahoClientFactory factory = new DefaultMqttPahoClientFactory();
        MqttConnectOptions options = new MqttConnectOptions();
        options.setServerURIs(new String[] { "tcp://host1:1883", "
tcp://host2:1883" });
        options.setUserName("username");
        options.setPassword("password".toCharArray());
        factory.setConnectionOptions(options);
        return factory;
    }

    @Bean
    @ServiceActivator(inputChannel = "mqttOutboundChannel")
    public MessageHandler mqttOutbound() {
        MqttPahoMessageHandler messageHandler =
                        new MqttPahoMessageHandler("testClient", mqttClientFactory
());
        messageHandler.setAsync(true);
        messageHandler.setDefaultTopic("testTopic");
        return messageHandler;
    }

    @Bean
    public MessageChannel mqttOutboundChannel() {
        return new DirectChannel();
    }

    @MessagingGateway(defaultRequestChannel = "mqttOutboundChannel")
    public interface MyGateway {

        void sendToMqtt(String data);

    }

}
```

27.2.2. Configuring with the Java DSL

The following Spring Boot application provides an example of configuring the outbound adapter with the Java DSL:

```
@SpringBootApplication
public class MqttJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(MqttJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public IntegrationFlow mqttOutboundFlow() {
        return f -> f.handle(new MqttPahoMessageHandler("tcp://host1:1883",
"someMqttClient"));
    }

}
```

27.3. Events

Certain application events are published by the adapters.

- `MqttConnectionFailedEvent` - published by both adapters if we fail to connect or a connection is subsequently lost.
- `MqttMessageSentEvent` - published by the outbound adapter when a message has been sent, if running in asynchronous mode.
- `MqttMessageDeliveredEvent` - published by the outbound adapter when the client indicates that a message has been delivered, if running in asynchronous mode.
- `MqttSubscribedEvent` - published by the inbound adapter after subscribing to the topics.

These events can be received by an `ApplicationListener<MqttIntegrationEvent>` or with an `@EventListener` method.

Chapter 28. Redis Support

Spring Integration 2.1 introduced support for Redis: “an open source advanced key-value store”. This support comes in the form of a Redis-based `MessageStore` as well as publish-subscribe messaging adapters that are supported by Redis through its `PUBLISH`, `SUBSCRIBE`, and `UNSUBSCRIBE` commands.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-redis</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-redis:5.3.0.RC1"
```

You also need to include Redis client dependency, e.g. Lettuce.

To download, install, and run Redis, see the Redis documentation.

28.1. Connecting to Redis

To begin interacting with Redis, you first need to connect to it. Spring Integration uses support provided by another Spring project, Spring Data Redis, which provides typical Spring constructs: `ConnectionFactory` and `Template`. Those abstractions simplify integration with several Redis client Java APIs. Currently Spring Data Redis supports Jedis and Lettuce.

28.1.1. Using `RedisConnectionFactory`

To connect to Redis, you can use one of the implementations of the `RedisConnectionFactory` interface. The following listing shows the interface definition:

```
public interface RedisConnectionFactory extends PersistenceExceptionTranslator {

    /**
     * Provides a suitable connection for interacting with Redis.
     * @return connection for interacting with Redis.
     */
    RedisConnection getConnection();
}
```

The following example shows how to create a `LettuceConnectionFactory` in Java:

```
LettuceConnectionFactory cf = new LettuceConnectionFactory();
cf.afterPropertiesSet();
```

The following example shows how to create a `LettuceConnectionFactory` in Spring's XML configuration:

```
<bean id="redisConnectionFactory"
    class="o.s.data.redis.connection.lettuce.LettuceConnectionFactory">
    <property name="port" value="7379" />
</bean>
```

The implementations of `RedisConnectionFactory` provide a set of properties, such as port and host, that you can set if needed. Once you have an instance of `RedisConnectionFactory`, you can create an instance of `RedisTemplate` and inject it with the `RedisConnectionFactory`.

28.1.2. Using `RedisTemplate`

As with other template classes in Spring (such as `JdbcTemplate` and `JmsTemplate`) `RedisTemplate` is a helper class that simplifies Redis data access code. For more information about `RedisTemplate` and its variations (such as `StringRedisTemplate`) see the Spring Data Redis documentation.

The following example shows how to create an instance of `RedisTemplate` in Java:

```
RedisTemplate rt = new RedisTemplate<String, Object>();
rt.setConnectionFactory(redisConnectionFactory);
```

The following example shows how to create an instance of `RedisTemplate` in Spring's XML configuration:

```
<bean id="redisTemplate"
          class="org.springframework.data.redis.core.RedisTemplate">
    <property name="connectionFactory" ref="redisConnectionFactory"/>
</bean>
```

28.2. Messaging with Redis

As mentioned in the introduction, Redis provides support for publish-subscribe messaging through its `PUBLISH`, `SUBSCRIBE`, and `UNSUBSCRIBE` commands. As with JMS and AMQP, Spring Integration provides message channels and adapters for sending and receiving messages through Redis.

28.2.1. Redis Publish/Subscribe channel

Similarly to JMS, there are cases where both the producer and consumer are intended to be part of the same application, running within the same process. You can accomplished this by using a pair of inbound and outbound channel adapters. However, as with Spring Integration's JMS support, there is a simpler way to address this use case. You can create a publish-subscribe channel, as the following example shows:

```
<int-redis:publish-subscribe-channel id="redisChannel" topic-name="si.test.topic
"/>
```

A `publish-subscribe-channel` behaves much like a normal `<publish-subscribe-channel/>` element from the main Spring Integration namespace. It can be referenced by both the `input-channel` and the `output-channel` attributes of any endpoint. The difference is that this channel is backed by a Redis topic name: a `String` value specified by the `topic-name` attribute. However, unlike JMS, this topic does not have to be created in advance or even auto-created by Redis. In Redis, topics are simple `String` values that play the role of an address. The producer and consumer can communicate by using the same `String` value as their topic name. A simple subscription to this channel means that asynchronous publish-subscribe messaging is possible between the producing and consuming endpoints. However, unlike the asynchronous message channels created by adding a `<queue/>` element within a simple Spring Integration `<channel/>` element, the messages are not stored in an in-memory queue. Instead, those messages are passed through Redis, which lets you rely on its support for persistence and clustering as well as its interoperability with other non-Java platforms.

28.2.2. Redis Inbound Channel Adapter

The Redis inbound channel adapter (`RedisInboundChannelAdapter`) adapts incoming Redis messages into Spring messages in the same way as other inbound adapters. It receives platform-specific messages (Redis in this case) and converts them to Spring messages by using a `MessageConverter` strategy. The following example shows how to configure a Redis inbound channel adapter:

```
<int-redis:inbound-channel-adapter id="redisAdapter"
       topics="thing1, thing2"
       channel="receiveChannel"
       error-channel="testErrorChannel"
       message-converter="testConverter" />

<bean id="redisConnectionFactory"
    class="o.s.data.redis.connection.lettuce.LettuceConnectionFactory">
    <property name="port" value="7379" />
</bean>

<bean id="testConverter" class="things.something.SampleMessageConverter" />
```

The preceding example shows a simple but complete configuration of a Redis inbound channel adapter. Note that the preceding configuration relies on the familiar Spring paradigm of auto-discovering certain beans. In this case, the `redisConnectionFactory` is implicitly injected into the adapter. You can specify it explicitly by using the `connection-factory` attribute instead.

Also, note that the preceding configuration injects the adapter with a custom `MessageConverter`. The approach is similar to JMS, where `MessageConverter` instances are used to convert between Redis messages and the Spring Integration message payloads. The default is a `SimpleMessageConverter`.

Inbound adapters can subscribe to multiple topic names, hence the comma-separated set of values in the `topics` attribute.

Since version 3.0, the inbound adapter, in addition to the existing `topics` attribute, now has the `topic-patterns` attribute. This attribute contains a comma-separated set of Redis topic patterns. For more information regarding Redis publish-subscribe, see Redis Pub/Sub.

Inbound adapters can use a `RedisSerializer` to deserialize the body of Redis messages. The `serializer` attribute of the `<int-redis:inbound-channel-adapter>` can be set to an empty string, which results in a `null` value for the `RedisSerializer` property. In this case, the raw `byte[]` bodies of Redis messages are provided as the message payloads.

Since version 5.0, you can provide an `Executor` instance to the inbound adapter by using the `task-executor` attribute of the `<int-redis:inbound-channel-adapter>`. Also, the received Spring Integration messages now have the `RedisHeaders.MESSAGE_SOURCE` header to indicate the source of the published message: topic or pattern. You can use this downstream for routing logic.

28.2.3. Redis Outbound Channel Adapter

The Redis outbound channel adapter adapts outgoing Spring Integration messages into Redis messages in the same way as other outbound adapters. It receives Spring Integration messages and converts them to platform-specific messages (Redis in this case) by using a `MessageConverter` strategy. The following example shows how to configure a Redis outbound channel adapter:

```
<int-redis:outbound-channel-adapter id="outboundAdapter"
    channel="sendChannel"
    topic="thing1"
    message-converter="testConverter"/>

<bean id="redisConnectionFactory"
    class="o.s.data.redis.connection.lettuce.LettuceConnectionFactory">
    <property name="port" value="7379"/>
</bean>

<bean id="testConverter" class="things.something.SampleMessageConverter" />
```

The configuration parallels the Redis inbound channel adapter. The adapter is implicitly injected with a `RedisConnectionFactory`, which is defined with `redisConnectionFactory` as its bean name. This example also includes the optional (and custom) `MessageConverter` (the `testConverter` bean).

Since Spring Integration 3.0, the `<int-redis:outbound-channel-adapter>` offers an alternative to the `topic` attribute: You can use the `topic-expression` attribute to determine the Redis topic for the message at runtime. These attributes are mutually exclusive.

28.2.4. Redis Queue Inbound Channel Adapter

Spring Integration 3.0 introduced a queue inbound channel adapter to “pop” messages from a Redis list. By default, it uses “right pop”, but you can configure it to use “left pop” instead. The adapter is message-driven. It uses an internal listener thread and does not use a poller.

The following listing shows all the available attributes for `queue-inbound-channel-adapter`:

```
<int-redis:queue-inbound-channel-adapter id=""  ①
                    channel=""  ②
                    auto-startup=""  ③
                    phase=""  ④
                    connection-factory=""  ⑤
                    queue=""  ⑥
                    error-channel=""  ⑦
                    serializer=""  ⑧
                    receive-timeout=""  ⑨
                    recovery-interval=""  ⑩
                    expect-message=""  ⑪
                    task-executor=""  ⑫
                    right-pop=""/>  ⑬
```

① The component bean name. If you do not provide the `channel` attribute, a `DirectChannel` is created and registered in the application context with this `id` attribute as the bean name. In this case, the endpoint itself is registered with the bean name `id` plus `.adapter`. (If the bean name were `thing1`, the endpoint is registered as `thing1.adapter`.)

② The `MessageChannel` to which to send `Message` instances from this Endpoint.

③ A `SmartLifecycle` attribute to specify whether this endpoint should start automatically after the application context start or not. It defaults to `true`.

④ A `SmartLifecycle` attribute to specify the phase in which this endpoint is started. It defaults to `0`.

⑤ A reference to a `RedisConnectionFactory` bean. It defaults to `redisConnectionFactory`.

⑥ The name of the Redis list on which the queue-based 'pop' operation is performed to get Redis messages.

⑦ The `MessageChannel` to which to send `ErrorMessage` instances when exceptions are received from the listening task of the endpoint. By default, the underlying `MessagePublishingErrorHandler` uses the default `errorChannel` from the application context.

⑧ The `RedisSerializer` bean reference. It can be an empty string, which means 'no serializer'. In this case, the raw `byte[]` from the inbound Redis message is sent to the `channel` as the `Message` payload. By default it is a `JdkSerializationRedisSerializer`.

⑨ The timeout in milliseconds for 'pop' operation to wait for a Redis message from the queue. The default is 1 second.

⑩ The time in milliseconds for which the listener task should sleep after exceptions on the 'pop' operation, before restarting the listener task.

⑪ Specifies whether this endpoint expects data from the Redis queue to contain entire `Message` instances. If this attribute is set to `true`, the `serializer` cannot be an empty string, because messages require some form of deserialization (JDK serialization by default). Its default is `false`.

⑫ A reference to a Spring `TaskExecutor` (or standard JDK 1.5+ `Executor`) bean. It is used for the underlying listening task. It defaults to a `SimpleAsyncTaskExecutor`.

⑬ Specifies whether this endpoint should use “right pop” (when `true`) or “left pop” (when `false`) to read messages from the Redis list. If `true`, the Redis List acts as a `FIFO` queue when used with a default Redis queue outbound channel adapter. Set it to `false` to use with software that writes to the list with “right push” or to achieve a stack-like message order. Its default is `true`. Since version 4.3.

The `task-executor` has to be configured with more than one thread for processing; otherwise there is a possible deadlock when the `RedisQueueMessageDrivenEndpoint` tries to restart the listener task after an error. The `errorChannel` can be used to process those errors, to avoid restarts, but it preferable to not expose your application to the possible deadlock situation. See Spring Framework Reference Manual for possible `TaskExecutor` implementations.

28.2.5. Redis Queue Outbound Channel Adapter

Spring Integration 3.0 introduced a queue outbound channel adapter to “push” to a Redis list from Spring Integration messages. By default, it uses “left push”, but you can configure it to use “right push” instead. The following listing shows all the available attributes for a Redis `queue-outbound-channel-adapter`:

```
<int-redis:queue-outbound-channel-adapter id=""  ①
                      channel=""  ②
                      connection-factory=""  ③
                      queue=""  ④
                      queue-expression=""  ⑤
                      serializer=""  ⑥
                      extract-payload=""  ⑦
                      left-push=""/>  ⑧
```

① The component bean name. If you do not provide the `channel` attribute, a `DirectChannel` is created and registered in the application context with this `id` attribute as the bean name. In this case, the endpoint is registered with a bean name of `id` plus `.adapter`. (If the bean name were `thing1`, the endpoint is registered as `thing1.adapter`.)

② The `MessageChannel` from which this endpoint receives `Message` instances.

③ A reference to a `RedisConnectionFactory` bean. It defaults to `redisConnectionFactory`.

④ The name of the Redis list on which the queue-based 'push' operation is performed to send Redis messages. This attribute is mutually exclusive with `queue-expression`.

⑤ A SpEL `Expression` to determine the name of the Redis list. It uses the incoming `Message` at runtime as the `#root` variable. This attribute is mutually exclusive with `queue`.

⑥ A `RedisSerializer` bean reference. It defaults to a `JdkSerializationRedisSerializer`. However, for `String` payloads, a `StringRedisSerializer` is used, if a `serializer` reference is not provided.

⑦ Specifies whether this endpoint should send only the payload or the entire `Message` to the Redis queue. It defaults to `true`.

⑧ Specifies whether this endpoint should use “left push” (when `true`) or “right push” (when `false`) to write messages to the Redis list. If `true`, the Redis list acts as a `FIFO` queue when used with a default Redis queue inbound channel adapter. Set it to `false` to use with software that reads from the list with “left pop” or to achieve a stack-like message order. It defaults to `true`. Since version 4.3.

28.2.6. Redis Application Events

Since Spring Integration 3.0, the Redis module provides an implementation of `IntegrationEvent`, which, in turn, is a `org.springframework.context.ApplicationEvent`. The `RedisExceptionEvent` encapsulates exceptions from Redis operations (with the endpoint being the “source” of the event). For example, the `<int-redis:queue-inbound-channel-adapter/>` emits those events after catching exceptions from the `BoundListOperations.rightPop` operation. The exception may be any generic `org.springframework.data.redis.RedisSystemException` or a `org.springframework.data.redis.RedisConnectionFailureException`. Handling these events with an `<int-event:inbound-channel-adapter/>` can be useful to determine problems with background Redis tasks and to take administrative actions.

28.3. Redis Message Store

As described in the *Enterprise Integration Patterns* (EIP) book, a message store lets you persist messages. This can be useful when dealing with components that have a capability to buffer messages (aggregator, resequencer, and others) when reliability is a concern. In Spring Integration, the `MessageStore` strategy also provides the foundation for the claim check pattern, which is described in EIP as well.

Spring Integration's Redis module provides the `RedisMessageStore`. The following example shows how to use it with a aggregator:

```
<bean id="redisMessageStore" class="o.s.i.redis.store.RedisMessageStore">
    <constructor-arg ref="redisConnectionFactory"/>
</bean>

<int:aggregator input-channel="inputChannel" output-channel="outputChannel"
         message-store="redisMessageStore"/>
```

The preceding example is a bean configuration, and it expects a `RedisConnectionFactory` as a constructor argument.

By default, the `RedisMessageStore` uses Java serialization to serialize the message. However, if you want to use a different serialization technique (such as JSON), you can provide your own serializer by setting the `valueSerializer` property of the `RedisMessageStore`.

Starting with version 4.3.10, the Framework provides Jackson serializer and deserializer implementations for `Message` instances and `MessageHeaders` instances — `MessageJacksonDeserializer` and `MessageHeadersJacksonSerializer`, respectively. They have to be configured with the `SimpleModule` options for the `ObjectMapper`. In addition, you should set `enableDefaultTyping` on the `ObjectMapper` to add type information for each serialized complex object. That type information is then used during deserialization. The framework provides a utility method called `JacksonJsonUtils.messagingAwareMapper()`, which is already supplied with all the previously mentioned properties and serializers. To manage JSON serialization in the `RedisMessageStore`, you must configure it in a fashion similar to the following example:

```
RedisMessageStore store = new RedisMessageStore(redisConnectionFactory);
ObjectMapper mapper = JacksonJsonUtils.messagingAwareMapper();
RedisSerializer<Object> serializer = new GenericJackson2JsonRedisSerializer(
mapper);
store.setValueSerializer(serializer);
```

Starting with version 4.3.12, `RedisMessageStore` supports the `prefix` option to allow distinguishing between instances of the store on the same Redis server.

28.3.1. Redis Channel Message Stores

The `RedisMessageStore` shown earlier maintains each group as a value under a single key (the group ID). While you can use this to back a `QueueChannel` for persistence, a specialized `RedisChannelMessageStore` is provided for that purpose (since version 4.0). This store uses a `LIST` for each channel, `LPUSH` when sending messages, and `RPOP` when receiving messages. By default, this store also uses JDK serialization, but you can modify the value serializer, as described earlier.

We recommend using this store backing channels, instead of using the general `RedisMessageStore`. The following example defines a Redis message store and uses it in a channel with a queue:

```
<bean id="redisMessageStore" class="o.s.i.redis.store.RedisChannelMessageStore">
    <constructor-arg ref="redisConnectionFactory"/>
</bean>

<int:channel id="somePersistentQueueChannel">
    <int:queue message-store="redisMessageStore"/>
<int:channel>
```

The keys used to store the data have the form: `<storeBeanName>:<channelId>` (in the preceding example, `redisMessageStore:somePersistentQueueChannel`).

In addition, a subclass `RedisChannelPriorityMessageStore` is also provided. When you use this with a `QueueChannel`, the messages are received in (FIFO) priority order. It uses the standard `IntegrationMessageHeaderAccessor.PRIORITY` header and supports priority values (`0 - 9`). Messages with other priorities (and messages with no priority) are retrieved in FIFO order after any messages with priority.

These stores implement only `BasicMessageGroupStore` and do not implement `MessageGroupStore`. They can be used only for situations such as backing a `QueueChannel`.

28.4. Redis Metadata Store

Spring Integration 3.0 introduced a new Redis-based `MetadataStore` (see Metadata Store) implementation. You can use the `RedisMetadataStore` to maintain the state of a `MetadataStore` across application restarts. You can use this new `MetadataStore` implementation with adapters such as:

- Feed
- File
- FTP
- SFTP

To instruct these adapters to use the new `RedisMetadataStore`, declare a Spring bean named `metadataStore`. The Feed inbound channel adapter and the feed inbound channel adapter both

automatically pick up and use the declared `RedisMetadataStore`. The following example shows how to declare such a bean:

```
<bean name="metadataStore" class="o.s.i.redis.store.metadata.RedisMetadataStore">
    <constructor-arg name="connectionFactory" ref="redisConnectionFactory"/>
</bean>
```

The `RedisMetadataStore` is backed by `RedisProperties`. Interaction with it uses `BoundHashOperations`, which, in turn, requires a `key` for the entire `Properties` store. In the case of the `MetadataStore`, this `key` plays the role of a region, which is useful in a distributed environment, when several applications use the same Redis server. By default, this `key` has a value of `MetaData`.

Starting with version 4.0, this store implements `ConcurrentMetadataStore`, letting it be reliably shared across multiple application instances where only one instance is allowed to store or modify a key's value.

You cannot use the `RedisMetadataStore.replace()` (for example, in the `AbstractPersistentAcceptOnceFileListFilter`) with a Redis cluster, since the `WATCH` command for atomicity is not currently supported.

28.5. Redis Store Inbound Channel Adapter

The Redis store inbound channel adapter is a polling consumer that reads data from a Redis collection and sends it as a `Message` payload. The following example shows how to configure a Redis store inbound channel adapter:

```
<int-redis:store-inbound-channel-adapter id="listAdapter"
    connection-factory="redisConnectionFactory"
    key="myCollection"
    channel="redisChannel"
    collection-type="LIST" >
    <int:poller fixed-rate="2000" max-messages-per-poll="10"/>
</int-redis:store-inbound-channel-adapter>
```

The preceding example shows how to configure a Redis store inbound channel adapter by using the `store-inbound-channel-adapter` element, providing values for various attributes, such as:

- `key` or `key-expression`: The name of the key for the collection being used.
- `collection-type`: An enumeration of the collection types supported by this adapter. The supported Collections are `LIST`, `SET`, `ZSET`, `PROPERTIES`, and `MAP`.
- `connection-factory`: Reference to an instance of `o.s.data.redis.connection.RedisConnectionFactory`.

- `redis-template`: Reference to an instance of `o.s.data.redis.core.RedisTemplate`.
- Other attributes that are common across all other inbound adapters (such as 'channel').

You cannot set both `redis-template` and `connection-factory`.

By default, the adapter uses a `StringRedisTemplate`. This uses `StringRedisSerializer` instances for keys, values, hash keys, and hash values. If your Redis store contains objects that are serialized with other techniques, you must supply a `RedisTemplate` configured with appropriate serializers. For example, if the store is written to using a Redis store outbound adapter that has its `extract-payload-elements` set to `false`, you must provide a `RedisTemplate` configured as follows:

```
<bean id="redisTemplate" class=
"org.springframework.data.redis.core.RedisTemplate">
    <property name="connectionFactory" ref="redisConnectionFactory
"/>
    <property name="keySerializer">
        <bean class=
"org.springframework.data.redis.serializer.StringRedisSerializer"/>
    </property>
    <property name="hashKeySerializer">
        <bean class=
"org.springframework.data.redis.serializer.StringRedisSerializer"/>
    </property>
</bean>
```

Th `RedisTemplate` uses `String` serializers for keys and hash keys and the default JDK Serialization serializers for values and hash values.

Because it has a literal value for the `key`, the preceding example is relatively simple and static. Sometimes, you may need to change the value of the key at runtime based on some condition. To do so, use `key-expression` instead, where the provided expression can be any valid SpEL expression.

Also, you may wish to perform some post-processing on the successfully processed data that was read from the Redis collection. For example, you may want to move or remove the value after its been processed. You can do so by using the transaction synchronization feature that was added with Spring Integration 2.2. The following example uses `key-expression` and transaction synchronization:

```
<int-redis:store-inbound-channel-adapter id=
"zsetAdapterWithSingleScoreAndSynchronization"
        connection-factory="redisConnectionFactory"
        key-expression="'presidents'"
        channel="otherRedisChannel"
        auto-startup="false"
        collection-type="ZSET">
            <int:poller fixed-rate="1000" max-messages-per-poll="2">
                <int:transactional synchronization-factory="syncFactory"/>
            </int:poller>
</int-redis:store-inbound-channel-adapter>

<int:transaction-synchronization-factory id="syncFactory">
    <int:after-commit expression="payload.removeByScore(18, 18)"/>
</int:transaction-synchronization-factory>

<bean id="transactionManager" class="o.s.i.transaction.PseudoTransactionManager"/>
```

You can declare your poller to be transactional by using a `transactional` element. This element can reference a real transaction manager (for example, if some other part of your flow invokes JDBC). If you do not have a "real" transaction, you can use an `o.s.i.transaction.PseudoTransactionManager`, which is an implementation of Spring's `PlatformTransactionManager` and enables the use of the transaction synchronization features of the Redis adapter when there is no actual transaction.

This does not make the Redis activities themselves transactional. It lets the synchronization of actions be taken before or after success (commit) or after failure (rollback).

Once your poller is transactional, you can set an instance of the `o.s.i.transaction.TransactionSynchronizationFactory` on the `transactional` element. `TransactionSynchronizationFactory` creates an instance of the `TransactionSynchronization`. For your convenience, we have exposed a default SpEL-based `TransactionSynchronizationFactory`, which lets you configure SpEL expressions, with their execution being coordinated (synchronized) with a transaction. Expressions for before-commit, after-commit, and after-rollback are supported, together with channels (one for each kind of event) where the evaluation result (if any) is sent. For each child element, you can specify `expression` and `channel` attributes. If only the `channel` attribute is present, the received message is sent there as part of the particular synchronization scenario. If only the `expression` attribute is present and the result of an expression is a non-null value, a message with the result as the payload is generated and sent to a default channel (`NullChannel`) and appears in the logs (at the `DEBUG` level). If you want the evaluation result to go to a specific channel, add a `channel` attribute. If the result of an expression is null or void, no message is generated.

For more information about transaction synchronization, see Transaction Synchronization.

28.6. RedisStore Outbound Channel Adapter

The RedisStore outbound channel adapter lets you write a message payload to a Redis collection, as the following example shows:

```
<int-redis:store-outbound-channel-adapter id="redisListAdapter"
          collection-type="LIST"
          channel="requestChannel"
          key="myCollection" />
```

The preceding configuration a Redis store outbound channel adapter by using the `store-inbound-channel-adapter` element. It provides values for various attributes, such as:

- `key` or `key-expression`: The name of the key for the collection being used.
- `extract-payload-elements`: If set to `true` (the default) and the payload is an instance of a "multi-value" object (that is, a `Collection` or a `Map`), it is stored by using "addAll" and "putAll" semantics. Otherwise, if set to `false`, the payload is stored as a single entry regardless of its type. If the payload is not an instance of a "multi-value" object, the value of this attribute is ignored and the payload is always stored as a single entry.
- `collection-type`: An enumeration of the `Collection` types supported by this adapter. The supported Collections are `LIST`, `SET`, `ZSET`, `PROPERTIES`, and `MAP`.
- `map-key-expression`: SpEL expression that returns the name of the key for the entry being stored. It applies only if the `collection-type` is `MAP` or `PROPERTIES` and 'extract-payload-elements' is false.
- `connection-factory`: Reference to an instance of `o.s.data.redis.connection.RedisConnectionFactory`.
- `redis-template`: Reference to an instance of `o.s.data.redis.core.RedisTemplate`.
- Other attributes that are common across all other inbound adapters (such as 'channel').

You cannot set both `redis-template` and `connection-factory`.

By default, the adapter uses a `StringRedisTemplate`. This uses `StringRedisSerializer` instances for keys, values, hash keys, and hash values. However, if `extract-payload-elements` is set to `false`, a `RedisTemplate` that has `StringRedisSerializer` instances for keys and hash keys and `JdkSerializationRedisSerializer` instances s for values and hash values will be used. With the JDK serializer, it is important to understand that Java serialization is used for all values, regardless of whether the value is actually a collection or not. If you need more control over the serialization of values, consider providing your own `RedisTemplate` rather than relying upon these defaults.

Because it has literal values for the `key` and other attributes, the preceding example is relatively simple and static. Sometimes, you may need to change the values dynamically at runtime based on some condition. To do so, use their `-expression` equivalents (`key-expression`, `map-key-expression`, and

so on), where the provided expression can be any valid SpEL expression.

28.7. Redis Outbound Command Gateway

Spring Integration 4.0 introduced the Redis command gateway to let you perform any standard Redis command by using the generic `RedisConnection#execute` method. The following listing shows the available attributes for the Redis outbound gateway:

```
<int-redis:outbound-gateway
        request-channel=""  ①
        reply-channel=""  ②
        requires-reply=""  ③
        reply-timeout=""  ④
        connection-factory=""  ⑤
        redis-template=""  ⑥
        arguments-serializer=""  ⑦
        command-expression=""  ⑧
        argument-expressions=""  ⑨
        use-command-variable=""  ⑩
        arguments-strategy="" /> ⑪
```

① The `MessageChannel` from which this endpoint receives `Message` instances.

② The `MessageChannel` where this endpoint sends reply `Message` instances.

③ Specifies whether this outbound gateway must return a non-null value. It defaults to `true`. A `ReplyRequiredException` is thrown when Redis returns a `null` value.

④ The timeout (in milliseconds) to wait until the reply message is sent. It is typically applied for queue-based limited reply-channels.

⑤ A reference to a `RedisConnectionFactory` bean. It defaults to `redisConnectionFactory`. It is mutually exclusive with 'redis-template' attribute.

⑥ A reference to a `RedisTemplate` bean. It is mutually exclusive with 'connection-factory' attribute.

⑦ A reference to an instance of `org.springframework.data.redis.serializer.RedisSerializer`. It is used to serialize each command argument to byte[], if necessary.

⑧ The SpEL expression that returns the command key. It defaults to the `redis_command` message header. It must not evaluate to `null`.

⑨ Comma-separated SpEL expressions that are evaluated as command arguments. Mutually exclusive with the `arguments-strategy` attribute. If you provide neither attribute, the `payload` is used as the command arguments. The argument expressions can evaluate to 'null' to support a variable number of arguments.

⑩ A `boolean` flag to specify whether the evaluated Redis command string is made available as the `#cmd` variable in the expression evaluation context in the `o.s.i.redis.outbound.ExpressionArgumentsStrategy` when `argument-expressions` is configured. Otherwise this attribute is ignored.

⑪ Reference to an instance of `o.s.i.redis.outbound.ArgumentsStrategy`. It is mutually exclusive with `argument-expressions` attribute. If you provide neither attribute, the `payload` is used as the command arguments.

You can use the `<int-redis:outbound-gateway>` as a common component to perform any desired Redis operation. For example, the following examlpe shows how to get incremented values from Redis atomic number:

```
<int-redis:outbound-gateway request-channel="requestChannel"
    reply-channel="replyChannel"
    command-expression="'INCR'"/>
```

The `Message` payload should have a name of `redisCounter`, which may be provided by `org.springframework.data.redis.support.atomic.RedisAtomicInteger` bean definition.

The `RedisConnection#execute` method has a generic `Object` as its return type. Real result depends on command type. For example, `MGET` returns a `List<byte[]>`. For more information about commands, their arguments and result type, see Redis Specification.

28.8. Redis Queue Outbound Gateway

Spring Integration introduced the Redis queue outbound gateway to perform request and reply scenarios. It pushes a conversation `UUID` to the provided `queue`, pushes the value with that `UUID` as its key to a Redis list, and waits for the reply from a Redis list with a key of `UUID' plus '.reply`. A different UUID is used for each interaction. The following listing shows the available attributes for a Redis outbound gateway:

```
<int-redis:queue-outbound-gateway
        request-channel=""  ①
        reply-channel=""  ②
        requires-reply=""  ③
        reply-timeout=""  ④
        connection-factory=""  ⑤
        queue=""  ⑥
        order=""  ⑦
        serializer=""  ⑧
        extract-payload=""/>  ⑨
```

① The `MessageChannel` from which this endpoint receives `Message` instances.

② The `MessageChannel` where this endpoint sends reply `Message` instances.

③ Specifies whether this outbound gateway must return a non-null value. This value is `false` by default. Otherwise, a `ReplyRequiredException` is thrown when Redis returns a `null` value.

④ The timeout (in milliseconds) to wait until the reply message is sent. It is typically applied for queue-based limited reply-channels.

⑤ A reference to a `RedisConnectionFactory` bean. It defaults to `redisConnectionFactory`. It is mutually exclusive with the 'redis-template' attribute.

⑥ The name of the Redis list to which the outbound gateway sends a conversation `UUID`.

⑦ The order of this outbound gateway when multiple gateways are registered.

⑧ The `RedisSerializer` bean reference. It can be an empty string, which means “no serializer”. In this case, the raw `byte[]` from the inbound Redis message is sent to the `channel` as the `Message` payload. By default, it is a `JdkSerializationRedisSerializer`.

⑨ Specifies whether this endpoint expects data from the Redis queue to contain entire `Message` instances. If this attribute is set to `true`, the `serializer` cannot be an empty string, because messages require some form of deserialization (JDK serialization by default).

28.9. Redis Queue Inbound Gateway

Spring Integration 4.1 introduced the Redis queue inbound gateway to perform request and reply scenarios. It pops a conversation `UUID` from the provided `queue`, pops the value with that `UUID` as its key from the Redis list, and pushes the reply to the Redis list with a key of `UUID` plus `.reply`. The following listing shows the available attributes for a Redis queue inbound gateway:

```
<int-redis:queue-inbound-gateway
        request-channel=""  ①
        reply-channel=""  ②
        executor=""  ③
        reply-timeout=""  ④
        connection-factory=""  ⑤
        queue=""  ⑥
        order=""  ⑦
        serializer=""  ⑧
        receive-timeout=""  ⑨
        expect-message=""  ⑩
        recovery-interval=""/>  ⑪
```

① The `MessageChannel` from which this endpoint receives `Message` instances.

② The `MessageChannel` where this endpoint sends reply `Message` instances.

③ A reference to a Spring `TaskExecutor` (or a standard JDK 1.5+ `Executor`) bean. It is used for the underlying listening task. It defaults to a `SimpleAsyncTaskExecutor`.

④ The timeout (in milliseconds) to wait until the reply message is sent. It is typically applied for queue-based limited reply-channels.

⑤ A reference to a `RedisConnectionFactory` bean. It defaults to `redisConnectionFactory`. It is mutually exclusive with 'redis-template' attribute.

⑥ The name of the Redis list for the conversation `UUID`.

⑦ The order of this inbound gateway when multiple gateways are registered.

⑧ The `RedisSerializer` bean reference. It can be an empty string, which means “no serializer”. In this case, the raw `byte[]` from the inbound Redis message is sent to the `channel` as the `Message` payload. It default to a `JdkSerializationRedisSerializer`. (Note that, in releases before version 4.3, it was a `StringRedisSerializer` by default. To restore that behavior, provide a reference to a `StringRedisSerializer`).

⑨ The timeout (in milliseconds) to wait until the receive message is fetched. It is typically applied for queue-based limited request-channels.

⑩ Specifies whether this endpoint expects data from the Redis queue to contain entire `Message` instances. If this attribute is set to `true`, the `serializer` cannot be an empty string, because messages require some form of deserialization (JDK serialization by default).

⑪ The time (in milliseconds) the listener task should sleep after exceptions on the “right pop” operation before restarting the listener task.

The `task-executor` has to be configured with more than one thread for processing; otherwise there is a possible deadlock when the `RedisQueueMessageDrivenEndpoint` tries to restart the listener task after an error. The `errorChannel` can be used to process those errors, to avoid restarts, but it preferable to not expose your application to the possible deadlock situation. See Spring Framework Reference Manual for possible `TaskExecutor` implementations.

28.10. Redis Lock Registry

Spring Integration 4.0 introduced the `RedisLockRegistry`. Certain components (for example, aggregator and resequencer) use a lock obtained from a `LockRegistry` instance to ensure that only one thread manipulates a group at a time. The `DefaultLockRegistry` performs this function within a single component. You can now configure an external lock registry on these components. When you use it with a shared `MessageGroupStore`, you can use the `RedisLockRegistry` to provide this functionality across multiple application instances, such that only one instance can manipulate the group at a time.

When a lock is released by a local thread, another local thread can generally acquire the lock immediately. If a lock is released by a thread using a different registry instance, it can take up to 100ms to acquire the lock.

To avoid “hung” locks (when a server fails), the locks in this registry are expired after a default 60 seconds, but you can configure this value on the registry. Locks are normally held for a much smaller time.

Because the keys can expire, an attempt to unlock an expired lock results in an exception being thrown. However, the resources protected by such a lock may have been compromised, so such exceptions should be considered to be severe. You should set the expiry at a large enough value to prevent this condition, but set it low enough that the lock can be recovered after a server failure in a reasonable amount of time.

Starting with version 5.0, the `RedisLockRegistry` implements `ExpirableLockRegistry`, which removes locks last acquired more than `age` ago and that are not currently locked.

Chapter 29. Resource Support

The resource inbound channel adapter builds upon Spring's `Resource` abstraction to support greater flexibility across a variety of actual types of underlying resources, such as a file, a URL, or a class path resource. Therefore, it is similar to but more generic than the file inbound channel adapter.

29.1. Resource Inbound Channel Adapter

The resource inbound channel adapter is a polling adapter that creates a `Message` whose payload is a collection of `Resource` objects.

`Resource` objects are resolved based on the pattern specified by the `pattern` attribute. The collection of resolved `Resource` objects is then sent as a payload within a `Message` to the adapter's channel. That is one major difference between resource inbound channel adapter and file inbound channel adapter: The latter buffers `File` objects and sends a single `File` object per `Message`.

The following example shows a simple configuration that finds all files that end with the 'properties' extension in the `things.thing1` package available on the classpath and sends them as the payload of a `Message` to the channel named 'resultChannel':

```
<int:resource-inbound-channel-adapter id="resourceAdapter"
                channel="resultChannel"
                pattern="classpath:things/thing1/*.properties">
    <int:poller fixed-rate="1000"/>
</int:resource-inbound-channel-adapter>
```

The resource inbound channel adapter relies on the `org.springframework.core.io.support.ResourcePatternResolver` strategy interface to resolve the provided pattern. It defaults to an instance of the current `ApplicationContext`. However, you can provide a reference to an instance of your own implementation of `ResourcePatternResolver` by setting the `pattern-resolver` attribute, as the following example shows:

```
<int:resource-inbound-channel-adapter id="resourceAdapter"
                channel="resultChannel"
                pattern="classpath:things/thing1/*.properties"
                pattern-resolver="myPatternResolver">
    <int:poller fixed-rate="1000"/>
</int:resource-inbound-channel-adapter>

<bean id="myPatternResolver" class="org.example.MyPatternResolver"/>
```

You may have a use case where you need to further filter the collection of resources resolved by the `ResourcePatternResolver`. For example, you may want to prevent resources that were already

resolved from appearing in a collection of resolved resources ever again. On the other hand, your resources might be updated rather often and you *do* want them to be picked up again. In other words, both defining an additional filter and disabling filtering altogether are valid use cases. You can provide your own implementation of the `org.springframework.integration.util.CollectionFilter` strategy interface, as the following example shows:

```
public interface CollectionFilter<T> {

    Collection<T> filter(Collection<T> unfilteredElements);

}
```

The `CollectionFilter` receives a collection of un-filtered elements (which are `Resource` objects in the preceding example), and it returns a collection of filtered elements of that same type.

If you define the adapter with XML but you do not specify a filter reference, the resource inbound channel adapter uses a default implementation of `CollectionFilter`. The implementation class of that default filter is `org.springframework.integration.util.AcceptOnceCollectionFilter`. It remembers the elements passed in the previous invocation in order to avoid returning those elements more than once.

To inject your own implementation of `CollectionFilter` instead, use the `filter` attribute, as the following example shows:

```
<int:resource-inbound-channel-adapter id="resourceAdapter"
                channel="resultChannel"
                pattern="classpath:things/thing1/*.properties"
                filter="myFilter">
    <int:poller fixed-rate="1000"/>
</int:resource-inbound-channel-adapter>

<bean id="myFilter" class="org.example.MyFilter"/>
```

If you do not need any filtering and want to disable even the default `CollectionFilter` strategy, provide an empty value for the filter attribute (for example, `filter=""`)

Chapter 30. RMI Support

This chapter explains how to use channel adapters that are specific to RMI (Remote Method Invocation) to distribute a system over multiple JVMs. The first section deals with sending messages over RMI. The second section shows how to receive messages over RMI. The last section shows how to define RMI channel adapters by using the namespace support.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-rmi</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-rmi:5.3.0.RC1"
```

30.1. Outbound RMI

To send messages from a channel over RMI, you can define an `RmiOutboundGateway`. This gateway uses Spring's `RmiProxyFactoryBean` internally to create a proxy for a remote gateway. Note that, to invoke a remote interface that does not use Spring Integration, you should use a service activator in combination with Spring's RmiProxyFactoryBean.

To configure the outbound gateway, you can write a bean definition similar the following:

```
<bean id="rmiOutGateway" class=org.spf.integration.rmi.RmiOutboundGateway>
    <constructor-arg value="rmi://host"/>
    <property name="replyChannel" value="replies"/>
</bean>
```

30.2. Inbound RMI

To receive messages over RMI, you need to use an `RmiInboundGateway`. You can configure the gateway as shown in the following example:

```
<bean id="rmiInGateway" class=org.spf.integration.rmi.RmiInboundGateway>
    <property name="requestChannel" value="requests"/>
</bean>
```

If you use an `errorChannel` on an inbound gateway, it the error flow normally returns a result or throws an exception. This is because it is likely that there is a corresponding outbound gateway waiting for a response of some kind. Consuming a message on the error flow and not replying results in no reply for the inbound gateway. Exceptions (on the main flow when there is no `errorChannel` or on the error flow) propagate to the corresponding inbound gateway.

30.3. RMI namespace support

To configure the inbound gateway, you can use the namespace support for it. The following code snippet shows the different configuration options that are supported:

```
<int-rmi:inbound-gateway id="gatewayWithDefaults" request-channel="testChannel"/>

<int-rmi:inbound-gateway id="gatewayWithCustomProperties" request-channel=
"testChannel"
    expect-reply="false" request-timeout="123" reply-timeout="456"/>

<int-rmi:inbound-gateway id="gatewayWithHost" request-channel="testChannel"
    registry-host="localhost"/>

<int-rmi:inbound-gateway id="gatewayWithPort" request-channel="testChannel"
    registry-port="1234" error-channel="rmiErrorChannel"/>

<int-rmi:inbound-gateway id="gatewayWithExecutorRef" request-channel="testChannel"
    remote-invocation-executor="invocationExecutor"/>
```

You can also use the namespace support to configure the outbound gateway. The following code snippet shows the different configuration for an outbound RMI gateway:

```
<int-rmi:outbound-gateway id="gateway"
    request-channel="localChannel"
    remote-channel="testChannel"
    host="localhost"/>
```

30.4. Configuring with Java Configuration

The following example shows how to configure an inbound gateway and an outbound gateway with Java:

```
@Bean
public RmiInboundGateway inbound() {
    RmiInboundGateway gateway = new RmiInboundGateway();
    gateway.setRequestChannel(requestChannel());
    gateway.setRegistryHost("host");
    gateway.setRegistryPort(port);
    return gateway;
}

@Bean
@ServiceActivator(inputChannel="inChannel")
public RmiOutboundGateway outbound() {
    RmiOutboundGateway gateway = new RmiOutboundGateway("rmi://host:port/"
        + RmiInboundGateway.SERVICE_NAME_PREFIX + "remoteChannelName");
    return gateway;
}
```

As of version 4.3, the outbound gateway has a second constructor that takes an `RmiProxyFactoryBeanConfigurer` instance, along with the service url argument. It allows further configuration before the proxy is created — for example, to inject a Spring Security `ContextPropagatingRemoteInvocationFactory`, as the following example shows:

```
@Bean
@ServiceActivator(inputChannel="inChannel")
public RmiOutboundGateway outbound() {
    RmiOutboundGateway gateway = new RmiOutboundGateway("rmi://host:port/"
                + RmiInboundGateway.SERVICE_NAME_PREFIX + "remoteChannelName",
        pfb -> {
            pfb.setRemoteInvocationFactory(new
ContextPropagatingRemoteInvocationFactory());
        });
    return gateway;
}
```

Starting with version 5.0, you can set this with the XML namespace by using the `configurer` attribute.

Chapter 31. RSocket Support

The RSocket Spring Integration module (`spring-integration-rsocket`) allows for executions of RSocket application protocol.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-rsocket</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-rsocket:5.3.0.RC1"
```

This module is available starting with version 5.2 and is based on the Spring Messaging foundation with its RSocket component implementations, such as `RSocketRequester`, `RSocketMessageHandler` and `RSocketStrategies`. See Spring Framework RSocket Support for more information about the RSocket protocol, terminology and components.

Before starting an integration flow processing via channel adapters, we need to establish an RSocket connection between server and client. For this purpose, Spring Integration RSocket support provides the `ServerRSocketConnector` and `ClientRSocketConnector` implementations of the `AbstractRSocketConnector`.

The `ServerRSocketConnector` exposes a listener on the host and port according to provided `io.rsocket.transport.ServerTransport` for accepting connections from clients. An internal `RSocketServer` instance can be customized with the `setServerConfigurer()`, as well as other options that can be configured, e.g. `RSocketStrategies` and `MimeType` for payload data and headers metadata. When a `setupRoute` is provided from the client requester (see `ClientRSocketConnector` below), a connected client is stored as a `RSocketRequester` under the key determined by the `clientRSocketKeyStrategy BiFunction<Map<String, Object>, DataBuffer, Object>`. By default a connect data is used for the key as a converted value to string with UTF-8 charset. Such an `RSocketRequester` registry can be used in the application logic to determine a particular client connection for interaction with it, or for publishing the same message to all connected clients. When a connection is established from the client, an `RSocketConnectedEvent` is emitted from the `ServerRSocketConnector`. This is similar to what is provided by the `@ConnectMapping` annotation in Spring Messaging module. The mapping pattern `*` means accept all the client routes. The `RSocketConnectedEvent` can be used to distinguish different routes via `DestinationPatternsMessageCondition.LOOKUP_DESTINATION_HEADER` header.

A typical server configuration might look like this:

```
@Bean
public RSocketStrategies rsocketStrategies() {
    return RSocketStrategies.builder()
        .decoder(StringDecoder.textPlainOnly())
        .encoder(CharSequenceEncoder.allMimeTypes())
        .dataBufferFactory(new DefaultDataBufferFactory(true))
        .build();
}

@Bean
public ServerRSocketConnector serverRSocketConnector() {
    ServerRSocketConnector serverRSocketConnector = new ServerRSocketConnector(
"localhost", 0);
    serverRSocketConnector.setRSocketStrategies(rsocketStrategies());
    serverRSocketConnector.setMetadataMimeType(new MimeType("message",
"x.rsocket.routing.v0"));
    serverRSocketConnector.setServerConfigurer((server) -> server.payloadDecoder
(PayloadDecoder.ZERO_COPY));
    serverRSocketConnector.setClientRSocketKeyStrategy((headers, data) -> ""
                                    + headers.get
(DestinationPatternsMessageCondition.LOOKUP_DESTINATION_HEADER));
    return serverRSocketConnector;
}

@EventListener
public void onApplicationEvent(RSocketConnectedEvent event) {
    ...
}
```

All the options, including `RSocketStrategies` bean and `@EventListener` for `RSocketConnectedEvent`, are optional. See `ServerRSocketConnector` JavaDocs for more information.

Starting with version 5.2.1, the `ServerRSocketMessageHandler` is extracted to a public, top-level class for possible connection with an existing RSocket server. When a `ServerRSocketConnector` is supplied with an external instance of `ServerRSocketMessageHandler`, it doesn't create an RSocket server internally and just delegates all the handling logic to the provided instance. In addition the `ServerRSocketMessageHandler` can be configured with a `messageMappingCompatible` flag to handle also `@MessageMapping` for an RSocket controller, fully replacing the functionality provided by the standard `RSocketMessageHandler`. This can be useful in mixed configurations, when classic `@MessageMapping` methods are present in the same application along with RSocket channel adapters and an externally configured RSocket server is present in the application.

The `ClientRSocketConnector` serves as a holder for `RSocketRequester` based on the `RSocket` connected via the provided `ClientTransport`. The `RSocketConnector` can be customized with the provided `RSocketConnectorConfigurer`. The `setupRoute` (with optional templates variables) and `setupData` with metadata can be also configured on this component.

A typical client configuration might look like this:

```
@Bean
public RSocketStrategies rsocketStrategies() {
    return RSocketStrategies.builder()
        .decoder(StringDecoder.textPlainOnly())
        .encoder(CharSequenceEncoder.allMimeTypes())
        .dataBufferFactory(new DefaultDataBufferFactory(true))
        .build();
}

@Bean
public ClientRSocketConnector clientRSocketConnector() {
    ClientRSocketConnector clientRSocketConnector =
            new ClientRSocketConnector("localhost", serverRSocketConnector()
.getBoundPort().block());
    clientRSocketConnector.setRSocketStrategies(rsocketStrategies());
    clientRSocketConnector.setSetupRoute("clientConnect/{user}");
    clientRSocketConnector.setSetupRouteVariables("myUser");
    return clientRSocketConnector;
}
```

Most of these options (including `RSocketStrategies` bean) are optional. Note how we connect to the locally started RSocket server on the arbitrary port. See `ServerRSocketConnector.clientRSocketKeyStrategy` for `setupData` use cases. Also see `ClientRSocketConnector` and its `AbstractRSocketConnector` superclass JavaDocs for more information.

Both `ClientRSocketConnector` and `ServerRSocketConnector` are responsible for mapping inbound channel adapters to their `path` configuration for routing incoming RSocket requests. See the next section for more information.

31.1. RSocket Inbound Gateway

The `RSocketInboundGateway` is responsible for receiving RSocket requests and producing responses (if any). It requires an array of `path` mapping which could be as patterns similar to MVC request mapping or `@MessageMapping` semantics. In addition (since version 5.2.2), a set of interaction models (see `RSocketInteractionModel`) can be configured on the `RSocketInboundGateway` to restrict RSocket requests to this endpoint by the particular frame type. By default all the interaction models are supported. Such a bean, according its `IntegrationRSocketEndpoint` implementation (extension of a `ReactiveMessageHandler`), is auto detected either by the `ServerRSocketConnector` or `ClientRSocketConnector` for a routing logic in the internal `IntegrationRSocketMessageHandler` for incoming requests. An `AbstractRSocketConnector` can be provided to the `RSocketInboundGateway` for explicit endpoint registration. This way, the auto-detection option is disabled on that `AbstractRSocketConnector`. The `RSocketStrategies` can also be injected into the `RSocketInboundGateway` or they are obtained from the provided `AbstractRSocketConnector` overriding any explicit injection. Decoders are used from those `RSocketStrategies` to decode a request payload according to the

provided requestElementType. If an RSocketPayloadReturnValueHandler.RESPONSE_HEADER header is not provided in incoming the Message, the RSocketInboundGateway treats a request as a fireAndForget RSocket interaction model. In this case, an RSocketInboundGateway performs a plain send operation into the outputChannel. Otherwise a MonoProcessor value from the RSocketPayloadReturnValueHandler.RESPONSE_HEADER header is used for sending a reply to the RSocket. For this purpose, an RSocketInboundGateway performs a sendAndReceiveMessageReactive operation on the outputChannel. The payload of the message to send downstream is always a Flux according to MessagingRSocket logic. When in a fireAndForget RSocket interaction model, the message has a plain converted payload. The reply payload could be a plain object or a Publisher - the RSocketInboundGateway converts both of them properly into an RSocket response according to the encoders provided in the RSocketStrategies.

Starting with version 5.3, a decodeFluxAsUnit option (default false) is added to the RSocketInboundGateway. By default incoming Flux is transformed the way that each its event is decoded separately. This is an exact behavior present currently with @MessageMapping semantics. To restore a previous behavior or decode the whole Flux as single unit according application requirements, the decodeFluxAsUnit has to be set to true. However the target decoding logic depends on the Decoder selected, e.g. a StringDecoder requires a new line separator (by default) to be present in the stream to indicate a byte buffer end.

See Configuring RSocket Endpoints with Java for samples how to configure an RSocketInboundGateway endpoint and deal with payloads downstream.

31.2. RSocket Outbound Gateway

The RSocketOutboundGateway is an AbstractReplyProducingMessageHandler to perform requests into RSocket and produce replies based on the RSocket replies (if any). A low level RSocket protocol interaction is delegated into an RSocketRequester resolved from the provided ClientRSocketConnector or from the RSocketRequesterMethodArgumentResolver.RSOCKET_REQUESTER_HEADER header in the request message on the server side. A target RSocketRequester on the server side can be resolved from an RSocketConnectedEvent or using ServerRSocketConnector.getClientRSocketRequester() API according some business key selected for connect request mappings via ServerRSocketConnector.setClientRSocketKeyStrategy(). See ServerRSocketConnector JavaDocs for more information.

The route to send request has to be configured explicitly (together with path variables) or via a SpEL expression which is evaluated against request message.

The RSocket interaction model can be provided via RSocketInteractionModel option or respective expression setting. By default a requestResponse is used for common gateway use-cases.

When request message payload is a Publisher, a publisherElementType option can be provided to encode its elements according an RSocketStrategies supplied in the target RSocketRequester. An expression for this option can evaluate to a ParameterizedTypeReference. See the RSocketRequester.RequestSpec.data() JavaDocs for more information about data and its type.

An RSocket request can also be enhanced with a metadata. For this purpose a metadataExpression against request message can be configured on the RSocketOutboundGateway. Such an expression must

evaluate to a `Map<Object, MimeType>`.

When `interactionModel` is not `fireAndForget`, an `expectedResponseType` must be supplied. It is a `String.class` by default. An expression for this option can evaluate to a `ParameterizedTypeReference`. See the `RSocketRequester.RetrieveSpec.retrieveMono()` and `RSocketRequester.RetrieveSpec.retrieveFlux()` JavaDocs for more information about reply data and its type.

A reply `payload` from the `RSocketOutboundGateway` is a `Mono` (even for a `fireAndForget` interaction model it is `Mono<Void>`) always making this component as `async`. Such a `Mono` is subscribed before producing into the `outputChannel` for regular channels or processed on demand by the `FluxMessageChannel`. A `Flux` response for the `requestStream` or `requestChannel` interaction model is also wrapped into a reply `Mono`. It can be flattened downstream by the `FluxMessageChannel` with a passthrough service activator:

```
@ServiceActivator(inputChannel = "rsocketReplyChannel", outputChannel =
"fluxMessageChannel")
public Flux<?> flattenRSocketResponse(Flux<?> payload) {
    return payload;
}
```

Or subscribed explicitly in the target application logic.

See Configuring RSocket Endpoints with Java for samples how to configure an `RSocketOutboundGateway` endpoint a deal with payloads downstream.

31.3. RSocket Namespace Support

Spring Integration provides an `rsocket` namespace and the corresponding schema definition. To include it in your configuration, add the following namespace declaration in your application context configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-rsocket="http://www.springframework.org/schema/integration/rsocket"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    https://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/rsocket
    https://www.springframework.org/schema/integration/rsocket/spring-integration-
rsocket.xsd">
    ...
</beans>
```

31.3.1. Inbound

To configure Spring Integration RSocket inbound channel adapters with XML, you need to use an appropriate `inbound-gateway` components from the `int-rsocket` namespace. The following example shows how to configure it:

```
<int-rsocket:inbound-gateway id="inboundGateway"
                             path="testPath"
                             interaction-models="requestStream,requestChannel"
                             rsocket-connector="clientRSocketConnector"
                             request-channel="requestChannel"
                             rsocket-strategies="rsocketStrategies"
                             request-element-type="byte[]"/>
```

A `ClientRSocketConnector` and `ServerRSocketConnector` should be configured as generic `<bean>` definitions.

31.3.2. Outbound

```
<int-rsocket:outbound-gateway id="outboundGateway"
                              client-rsocket-connector="clientRSocketConnector"
                              auto-startup="false"
                              interaction-model="fireAndForget"
                              route-expression="'testRoute'"
                              request-channel="requestChannel"
                              publisher-element-type="byte[]"
                              expected-response-type="java.util.Date"
                              metadata-expression="{'metadata': new
org.springframework.util.MimeType('*')}"/>
```

See `spring-integration-rsocket.xsd` for description for all those XML attributes.

31.4. Configuring RSocket Endpoints with Java

The following example shows how to configure an RSocket inbound endpoint with Java:

```
@Bean
public RSocketInboundGateway rsocketInboundGatewayRequestReply() {
    RSocketInboundGateway rsocketInboundGateway = new RSocketInboundGateway("echo
");
    rsocketInboundGateway.setRequestChannelName("requestReplyChannel");
    return rsocketInboundGateway;
}

@Transformer(inputChannel = "requestReplyChannel")
public Mono<String> echoTransformation(Flux<String> payload) {
    return payload.next().map(String::toUpperCase);
}
```

A `ClientRSocketConnector` or `ServerRSocketConnector` is assumed in this configuration with meaning for auto-detection of such an endpoint on the “echo” path. Pay attention to the `@Transformer` signature with its fully reactive processing of the RSocket requests and producing reactive replies.

The following example shows how to configure a RSocket inbound gateway with the Java DSL:

```
@Bean
public IntegrationFlow rsocketUpperCaseFlow() {
    return IntegrationFlows
        .from(RSockets.inboundGateway("/uppercase")
                  .interactionModels(RSocketInteractionModel.requestChannel))
        .<Flux<String>, Mono<String>>transform((flux) -> flux.next().map(String:
:toUpperCase))
        .get();
}
```

A `ClientRSocketConnector` or `ServerRSocketConnector` is assumed in this configuration with meaning for auto-detection of such an endpoint on the "/uppercase" path and expected interaction model as "request channel".

The following example shows how to configure a RSocket outbound gateway with Java:

```
@Bean
@ServiceActivator(inputChannel = "requestChannel", outputChannel = "replyChannel")
public RSocketOutboundGateway rsocketOutboundGateway() {
    RSocketOutboundGateway rsocketOutboundGateway =
            new RSocketOutboundGateway(
                    new FunctionExpression<Message<?>>((m) ->
                        m.getHeaders().get("route_header")));
    rsocketOutboundGateway.setInteractionModelExpression(
            new FunctionExpression<Message<?>>((m) -> m.getHeaders().get(
"rsocket_interaction_model")));
    rsocketOutboundGateway.setClientRSocketConnector(clientRSocketConnector());
    return rsocketOutboundGateway;
}
```

The `setClientRSocketConnector()` is required only for the client side. On the server side, the `RSocketRequesterMethodArgumentResolver.RSOCKET_REQUESTER_HEADER` header with an `RSocketRequester` value must be supplied in the request message.

The following example shows how to configure a RSocket outbound gateway with the Java DSL:

```
@Bean
public IntegrationFlow rsocketUpperCaseRequestFlow(ClientRSocketConnector
clientRSocketConnector) {
    return IntegrationFlows
        .from(Function.class)
        .handle(RSockets.outboundGateway("/uppercase")
            .interactionModel(RSocketInteractionModel.requestResponse)
            .expectedResponseType(String.class)
            .clientRSocketConnector(clientRSocketConnector))
        .get();
}
```

See `IntegrationFlow` as a Gateway for more information how to use a mentioned `Function` interface in the beginning of the flow above.

Chapter 32. SFTP Adapters

Spring Integration provides support for file transfer operations over SFTP.

The Secure File Transfer Protocol (SFTP) is a network protocol that lets you transfer files between two computers on the Internet over any reliable stream.

The SFTP protocol requires a secure channel, such as SSH, and visibility to a client's identity throughout the SFTP session.

Spring Integration supports sending and receiving files over SFTP by providing three client side endpoints: inbound channel adapter, outbound channel adapter, and outbound gateway. It also provides convenient namespace configuration to define these client components.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-sftp</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-sftp:5.3.0.RC1"
```

To include the SFTP namespace in your xml configuration, include the following attributes on the root element:

```
xmlns:int-sftp="http://www.springframework.org/schema/integration/sftp"
xsi:schemaLocation="http://www.springframework.org/schema/integration/sftp
    https://www.springframework.org/schema/integration/sftp/spring-integration-
sftp.xsd"
```

32.1. SFTP Session Factory

As of version 3.0, sessions are no longer cached by default. See SFTP Session Caching.

Before configuring SFTP adapters, you must configure an SFTP session factory. You can configure the SFTP session factory with a regular bean definition, as the following example shows:

```
<beans:bean id="sftpSessionFactory"
    class="org.springframework.integration.sftp.session.DefaultSftpSessionFactory
">
    <beans:property name="host" value="localhost"/>
    <beans:property name="privateKey" value="classpath:META-INF/keys/sftpTest"/>
    <beans:property name="privateKeyPassphrase" value="springIntegration"/>
    <beans:property name="port" value="22"/>
    <beans:property name="user" value="kermit"/>
</beans:bean>
```

Every time an adapter requests a session object from its `SessionFactory`, a new SFTP session is created. Under the covers, the SFTP Session Factory relies on the JSch library to provide the SFTP capabilities.

However, Spring Integration also supports the caching of SFTP sessions. See SFTP Session Caching for more information.

JSch supports multiple channels (operations) over a connection to the server. By default, the Spring Integration session factory uses a separate physical connection for each channel. Since Spring Integration 3.0, you can configure the session factory (using a boolean constructor arg - default `false`) to use a single connection to the server and create multiple `JSch` channels on that single connection.

When using this feature, you must wrap the session factory in a caching session factory, as described later, so that the connection is not physically closed when an operation completes.

If the cache is reset, the session is disconnected only when the last channel is closed.

The connection is refreshed if it is found to be disconnected when a new operation obtains a session.

If you experience connectivity problems and would like to trace session creation and see which sessions are polled, you may enable tracing by setting the logger to `TRACE` level (for example, `log4j.category.org.springframework.integration.sftp=TRACE`). See SFTP/JSCH Logging.

Now all you need to do is inject this SFTP session factory into your adapters.

A more practical way to provide values for the SFTP session factory is to use Spring's property placeholder support.

32.1.1. Configuration Properties

The following list describes all the properties that are exposed by the `DefaultSftpSessionFactory`.

`isSharedSession` (constructor argument)::When `true`, a single connection is used, and `JSch Channels` are multiplexed. It defaults to `false`.

`clientVersion`::Lets you set the client version property. It's default depends on the underlying JSch version but it will look like: *SSH-2.0-JSCH-0.1.45*

`enableDaemonThread`::If `true`, all threads are daemon threads. If set to `false`, normal non-daemon threads are used instead. This property is set on the underlying session. There, this property defaults to `false`.

`host`::The URL of the host to which you want to connect. Required.

`hostKeyAlias`::Sets the host key alias, which is used when comparing the host key to the known hosts list.

`knownHostsResource`::Specifies the file resource that used for a host key repository. The file has the same format as OpenSSH's `known_hosts` file and is required and must be pre-populated if `allowUnknownKeys` is false.

`password`::The password to authenticate against the remote host. If a password is not provided, then the `privateKey` property is required. It is not allowed if you set `userInfo`. The password is obtained from that object.

`port`::The port over which the SFTP connection shall be established. If not specified, this value defaults to `22`. If specified, this properties must be a positive number.

`privateKey`::Lets you set a resource that represents the location of the private key used for authenticating against the remote host. If the `privateKey` is not provided, then the `password` property is required.

`privateKeyPassphrase`::The password for the private key. If you set `userInfo`, `privateKeyPassphrase` is not allowed . The passphrase is obtained from that object. Optional.

`proxy`::Allows for specifying a JSch-based proxy. If set, the proxy object is used to create the connection to the remote host through the proxy. See Proxy Factory Bean for a convenient way to configure the proxy.

`serverAliveCountMax`::Specifies the number of server-alive messages, which are sent without any reply from the server before disconnecting. If not set, this property defaults to `1`.

`serverAliveInterval`::Sets the timeout interval (in milliseconds) before a server-alive message is sent, in case no message is received from the server.

`sessionConfig`::By using `Properties`, you can set additional configuration setting on the underlying JSch Session.

`socketFactory`::Lets you pass in a `SocketFactory`. The socket factory is used to create a socket to the

target host. When a proxy is used, the socket factory is passed to the proxy. By default, plain TCP sockets are used.

`timeout`::The timeout property is used as the socket timeout parameter, as well as the default connection timeout. Defaults to `0`, which means, that no timeout will occur.

`user`::The remote user to use. Required.

`allowUnknownKeys`::Set to `true` to allow connections to hosts with unknown (or changed) keys. Its default is 'false'. It is applied only if no `userInfo` is provided. If `false`, a pre-populated `knownHosts` file is required.

`userInfo`::Set a custom `UserInfo` to be used during authentication. In particular, `promptYesNo()` is invoked when an unknown (or changed) host key is received. See also `allowUnknownKeys`. When you provide a `UserInfo`, the `password` and private key `passphrase` are obtained from it, and you cannot set discrete `password` and `privateKeyPassphrase` properties.

32.2. Proxy Factory Bean

`Jsch` provides a mechanism to connect to the server over an HTTP or SOCKS proxy. To use this feature, configure the `Proxy` and provide a reference to the `DefaultSftpSessionFactory`, as discussed earlier. Three implementations are provided by `Jsch`: `HTTP`, `SOCKS4`, and `SOCKS5`. Spring Integration 4.3 introduced a `FactoryBean`, easing configuration of these proxies by allowing property injection, as the following example shows:

```
<bean id="proxySocks5" class=
"org.springframework.integration.sftp.session.JschProxyFactoryBean">
    <constructor-arg value="SOCKS5" />
    <constructor-arg value="${sftp.proxy.address}" />
    <constructor-arg value="${sftp.proxy.port}" />
    <constructor-arg value="${sftp.proxy.user}" />
    <constructor-arg value="${sftp.proxy.pw}" />
</bean>

<bean id="sessionFactory"
          class=
"org.springframework.integration.sftp.session.DefaultSftpSessionFactory" >
    ...
    <property name="proxy" ref="proxySocks5" />
    ...
</bean>
```

32.3. Delegating Session Factory

Version 4.2 introduced the `DelegatingSessionFactory`, which allows the selection of the actual session factory at runtime. Prior to invoking the SFTP endpoint, you can call `setThreadKey()` on the

factory to associate a key with the current thread. That key is then used to look up the actual session factory to be used. You can clear the key by calling `clearThreadKey()` after use.

We added convenience methods so that you can more easily do so from a message flow, as the following example shows:

```
<bean id="dsf" class=
"org.springframework.integration.file.remote.session.DelegatingSessionFactory">
    <constructor-arg>
        <bean class="o.s.i.file.remote.session.DefaultSessionFactoryLocator">
            <!-- delegate factories here -->
        </bean>
    </constructor-arg>
</bean>

<int:service-activator input-channel="in" output-channel="c1"
        expression="@dsf.setThreadKey(#root, headers['factoryToUse'])" />

<int-sftp:outbound-gateway request-channel="c1" reply-channel="c2" ... />

<int:service-activator input-channel="c2" output-channel="out"
        expression="@dsf.clearThreadKey(#root)" />
```

When using session caching (see SFTP Session Caching), each of the delegates should be cached. You cannot cache the `DelegatingSessionFactory` itself.

Starting with version 5.0.7, the `DelegatingSessionFactory` can be used in conjunction with a `RotatingServerAdvice` to poll multiple servers; see Inbound Channel Adapters: Polling Multiple Servers and Directories.

32.4. SFTP Session Caching

Starting with Spring Integration version 3.0, sessions are no longer cached by default. The `cache-sessions` attribute is no longer supported on endpoints. If you wish to cache sessions, you must use a `CachingSessionFactory` (see the next example).

In versions prior to 3.0, the sessions were automatically cached by default. A `cache-sessions` attribute was available for disabling the auto caching, but that solution did not provide a way to configure other session-caching attributes. For example, you could not limit on the number of sessions created. To support that requirement and other configuration options, we added a `CachingSessionFactory`. It provides `sessionCacheSize` and `sessionWaitTimeout` properties. As its name suggests, the `sessionCacheSize` property controls how many active sessions the factory maintains in its cache (the default is unbounded). If the `sessionCacheSize` threshold has been reached, any attempt to acquire another session blocks until either one of the cached sessions becomes available or until the wait time for a session expires (the default wait time is `Integer.MAX_VALUE`). The

`sessionWaitTimeout` property enables configuration of the wait time.

If you want your sessions to be cached, configure your default session factory (as described earlier) and then wrap it in an instance of `CachingSessionFactory` where you may provide those additional properties. The following example shows how to do so:

```
<bean id="sftpSessionFactory"
    class="org.springframework.integration.sftp.session.DefaultSftpSessionFactory">
    <property name="host" value="localhost"/>
</bean>

<bean id="cachingSessionFactory"
    class="org.springframework.integration.file.remote.session.CachingSessionFactory">
    <constructor-arg ref="sftpSessionFactory"/>
    <constructor-arg value="10"/>
    <property name="sessionWaitTimeout" value="1000"/>
</bean>
```

The preceding example creates a `CachingSessionFactory` with its `sessionCacheSize` set to `10` and its `sessionWaitTimeout` set to one second (1000 milliseconds).

Starting with Spring Integration version 3.0, the `CachingConnectionFactory` provides a `resetCache()` method. When invoked, all idle sessions are immediately closed and in-use sessions are closed when they are returned to the cache. When using `isSharedSession=true`, the channel is closed and the shared session is closed only when the last channel is closed. New requests for sessions establish new sessions as necessary.

Starting with version 5.1, the `CachingSessionFactory` has a new property `testSession`. When true, the session will be tested by performing a `stat(getHome())` command to ensure it is still active; if not, it will be removed from the cache; a new session is created if no active sessions are in the cache.

32.5. Using `RemoteFileTemplate`

Spring Integration version 3.0 provides a new abstraction over the `SftpSession` object. The template provides methods to send, retrieve (as an `InputStream`), remove, and rename files. In addition, we provide an `execute` method to let the caller run multiple operations on the session. In all cases, the template takes care of reliably closing the session. For more information, see the Javadoc for `RemoteFileTemplate` There is a subclass for SFTP: `SftpRemoteFileTemplate`.

We added additional methods in version 4.1, including `getClientInstance()`. It provides access to the underlying `ChannelSftp`, which enables access to low-level APIs.

Version 5.0 introduced the `RemoteFileOperations.invoke(OperationsCallback<F, T> action)` method. This method lets several `RemoteFileOperations` calls be called in the scope of the same thread-bounded `Session`. This is useful when you need to perform several high-level operations of the `RemoteFileTemplate` as one unit of work. For example, `AbstractRemoteFileOutboundGateway` uses it with the `mput` command implementation, where we perform a `put` operation for each file in the provided directory and recursively for its sub-directories. See the Javadoc for more information.

32.6. SFTP Inbound Channel Adapter

The SFTP inbound channel adapter is a special listener that connects to the server and listens for the remote directory events (such as a new file being created), at which point it initiates a file transfer. The following example shows how to configure an SFTP inbound channel adapter:

```
<int-sftp:inbound-channel-adapter id="sftpAdapterAutoCreate"
              session-factory="sftpSessionFactory"
            channel="requestChannel"
            filename-pattern="*.txt"
            remote-directory="/foo/bar"
            preserve-timestamp="true"
            local-directory="file:target/foo"
            auto-create-local-directory="true"
            local-filename-generator-expression="#this.toUpperCase() + '.a'"
            scanner="myDirScanner"
            local-filter="myFilter"
            temporary-file-suffix=".writing"
            max-fetch-size="-1"
            delete-remote-files="false">
        <int:poller fixed-rate="1000"/>
</int-sftp:inbound-channel-adapter>
```

The preceding configuration example shows how to provide values for various attributes, including the following:

- `local-directory`: The location to which files are going to be transferred
- `remote-directory`: The remote source directory from which files are going to be transferred
- `session-factory`: A reference to the bean we configured earlier

By default, the transferred file carries the same name as the original file. If you want to override this behavior, you can set the `local-filename-generator-expression` attribute, which lets you provide a SpEL expression to generate the name of the local file. Unlike outbound gateways and adapters, where the root object of the SpEL evaluation context is a `Message`, this inbound adapter does not yet have the message at the time of evaluation, since that is what it ultimately generates with the transferred file as its payload. Consequently, the root object of the SpEL evaluation context is the original name of the remote file (a `String`).

The inbound channel adapter first retrieves the file to a local directory and then emits each file according to the poller configuration. Starting with version 5.0, you can limit the number of files fetched from the SFTP server when new file retrievals are needed. This can be beneficial when the target files are large or when running in a clustered system with a persistent file list filter, discussed later in this section. Use `max-fetch-size` for this purpose. A negative value (the default) means no limit and all matching files are retrieved. See Inbound Channel Adapters: Controlling Remote File Fetching for more information. Since version 5.0, you can also provide a custom `DirectoryScanner` implementation to the `inbound-channel-adapter` by setting the `scanner` attribute.

Starting with Spring Integration 3.0, you can specify the `preserve-timestamp` attribute (the default is `false`). When `true`, the local file's modified timestamp is set to the value retrieved from the server. Otherwise, it is set to the current time.

Starting with version 4.2, you can specify `remote-directory-expression` instead of `remote-directory`, which lets you dynamically determine the directory on each poll — for example, `remote-directory-expression="@myBean.determineRemoteDir()"`.

Sometimes, file filtering based on the simple pattern specified via `filename-pattern` attribute might not suffice. If this is the case, you can use the `filename-regex` attribute to specify a regular expression (for example, `filename-regex=".*\.test$"`). If you need complete control, you can use the `filter` attribute to provide a reference to a custom implementation of the `org.springframework.integration.file.filters.FileListFilter`, which is a strategy interface for filtering a list of files. This filter determines which remote files are retrieved. You can also combine a pattern-based filter with other filters (such as an `AcceptOnceFileListFilter`, to avoid synchronizing files that have previously been fetched) by using a `CompositeFileListFilter`.

The `AcceptOnceFileListFilter` stores its state in memory. If you wish the state to survive a system restart, consider using the `SftpPersistentAcceptOnceFileListFilter` instead. This filter stores the accepted file names in an instance of the `MetadataStore` strategy (see Metadata Store). This filter matches on the filename and the remote modified time.

Since version 4.0, this filter requires a `ConcurrentMetadataStore`. When used with a shared data store (such as `Redis` with the `RedisMetadataStore`), this lets filter keys be shared across multiple application or server instances.

Starting with version 5.0, the `SftpPersistentAcceptOnceFileListFilter` with an in-memory `SimpleMetadataStore` is applied by default for the `SftpInboundFileSynchronizer`. This filter is also applied, together with the `regex` or `pattern` option in the XML configuration, as well as through `SftpInboundChannelAdapterSpec` in Java DSL. You can handle any other use-cases by using `CompositeFileListFilter` (or `ChainFileListFilter`).

The above discussion refers to filtering the files before retrieving them. Once the files have been retrieved, an additional filter is applied to the files on the file system. By default, this is an `AcceptOnceFileListFilter`, which, as discussed in this section, retains state in memory and does not consider the file's modified time. Unless your application removes files after processing, the adapter re-processes the files on disk by default after an application restart.

Also, if you configure the `filter` to use a `SftpPersistentAcceptOnceFileListFilter` and the remote file timestamp changes (causing it to be re-fetched), the default local filter does not allow this new file to be processed.

For more information about this filter, and how it is used, see Remote Persistent File List Filters.

You can use the `local-filter` attribute to configure the behavior of the local file system filter. Starting with version 4.3.8, a `FileSystemPersistentAcceptOnceFileListFilter` is configured by default. This filter stores the accepted file names and modified timestamp in an instance of the `MetadataStore` strategy (see Metadata Store) and detects changes to the local file modified time. The default `MetadataStore` is a `SimpleMetadataStore` that stores state in memory.

Since version 4.1.5, these filters have a new property called `flushOnUpdate`, which causes them to flush the metadata store on every update (if the store implements `Flushable`).

Further, if you use a distributed `MetadataStore` (such as Redis Metadata Store or Gemfire Metadata Store), you can have multiple instances of the same adapter or application and be sure that one and only one instance processes a file.

The actual local filter is a `CompositeFileListFilter` that contains the supplied filter and a pattern filter that prevents processing files that are in the process of being downloaded (based on the `temporary-file-suffix`). Files are downloaded with this suffix (the default is `.writing`), and the files are renamed to their final names when the transfer is complete, making them 'visible' to the filter.

See the schema for more detail on these attributes.

SFTP inbound channel adapter is a polling consumer. Therefore, you must configure a poller (either a global default or a local element). Once the file has been transferred to a local directory, a message with `java.io.File` as its payload type is generated and sent to the channel identified by the `channel` attribute.

32.6.1. More on File Filtering and Large Files

Sometimes, a file that just appeared in the monitored (remote) directory is not complete. Typically such a file is written with some temporary extension (such as `.writing` on a file named `something.txt.writing`) and then renamed after the writing process completes. In most cases, developers are interested only in files that are complete and would like to filter only those files. To handle these scenarios, you can use the filtering support provided by the `filename-pattern`, `filename-regex`, and `filter` attributes. If you need a custom filter implementation, you can include a reference in your adapter by setting the `filter` attribute. The following example shows how to do so:

```
<int-sftp:inbound-channel-adapter id="sftpInbondAdapter"
            channel="receiveChannel"
            session-factory="sftpSessionFactory"
            filter="customFilter"
            local-directory="file:/local-test-dir"
            remote-directory="/remote-test-dir">
        <int:poller fixed-rate="1000" max-messages-per-poll="10" task-executor=
"executor"/>
</int-sftp:inbound-channel-adapter>

<bean id="customFilter" class="org.foo.CustomFilter"/>
```

32.6.2. Recovering from Failures

You should understand the architecture of the adapter. A file synchronizer fetches the files, and a `FileReadingMessageSource` emits a message for each synchronized file. As discussed earlier, two

filters are involved. The `filter` attribute (and patterns) refers to the remote (SFTP) file list, to avoid fetching files that have already been fetched. the `FileReadingMessageSource` uses the `local-filter` to determine which files are to be sent as messages.

The synchronizer lists the remote files and consults its filter. The files are then transferred. If an IO error occurs during file transfer, any files that have already been added to the filter are removed so that they are eligible to be re-fetched on the next poll. This applies only if the filter implements `ReversibleFileListFilter` (such as the `AcceptOnceFileListFilter`).

If, after synchronizing the files, an error occurs on the downstream flow processing a file, no automatic rollback of the filter occurs, so the failed file is not reprocessed by default.

If you wish to reprocess such files after a failure, you can use a configuration similar to the following to facilitate the removal of the failed file from the filter:

```
<int-sftp:inbound-channel-adapter id="sftpAdapter"
        session-factory="sftpSessionFactory"
        channel="requestChannel"
        remote-directory-expression="'/sftpSource'"
        local-directory="file:myLocalDir"
        auto-create-local-directory="true"
        filename-pattern="*.txt">
    <int:poller fixed-rate="1000">
        <int:transactional synchronization-factory="syncFactory" />
    </int:poller>
</int-sftp:inbound-channel-adapter>

<bean id="acceptOnceFilter"
    class="org.springframework.integration.file.filters.AcceptOnceFileListFilter"
/>

<int:transaction-synchronization-factory id="syncFactory">
    <int:after-rollback expression="payload.delete()" />
</int:transaction-synchronization-factory>

<bean id="transactionManager"
    class="org.springframework.integration.transaction.PseudoTransactionManager"
/>
```

The preceding configuration works for any `ResettableFileListFilter`.

Starting with version 5.0, the inbound channel adapter can build sub-directories locally, according to the generated local file name. That can be a remote sub-path as well. To be able to read a local directory recursively for modification according to the hierarchy support, you can now supply an internal `FileReadingMessageSource` with a new `RecursiveDirectoryScanner` based on the `Files.walk()` algorithm. See `AbstractInboundFileSynchronizingMessageSource.setScanner()` for more information. Also, you can now switch the `AbstractInboundFileSynchronizingMessageSource` to the `WatchService` -based `DirectoryScanner` by using `setUseWatchService()` option. It is also configured for all the

`WatchEventType` instances to react for any modifications in local directory. The reprocessing sample shown earlier is based on the built-in functionality of the `FileReadingMessageSource.WatchServiceDirectoryScanner`, which uses `ResettableFileListFilter.remove()` when the file is deleted (`StandardWatchEventKinds.ENTRY_DELETE`) from the local directory. See `WatchServiceDirectoryScanner` for more information.

32.6.3. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the inbound adapter with Java:

```
@SpringBootApplication
public class SftpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(SftpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public SessionFactory<LsEntry> sftpSessionFactory() {
        DefaultSftpSessionFactory factory = new DefaultSftpSessionFactory(true);
        factory.setHost("localhost");
        factory.setPort(port);
        factory.setUser("foo");
        factory.setPassword("foo");
        factory.setAllowUnknownKeys(true);
        factory.setTestSession(true);
        return new CachingSessionFactory<LsEntry>(factory);
    }

    @Bean
    public SftpInboundFileSynchronizer sftpInboundFileSynchronizer() {
        SftpInboundFileSynchronizer fileSynchronizer = new
SftpInboundFileSynchronizer(sftpSessionFactory());
        fileSynchronizer.setDeleteRemoteFiles(false);
        fileSynchronizer.setRemoteDirectory("foo");
        fileSynchronizer.setFilter(new SftpSimplePatternFileListFilter("*.xml"));
        return fileSynchronizer;
    }

    @Bean
    @InboundChannelAdapter(channel = "sftpChannel", poller = @Poller(fixedDelay =
"5000"))
    public MessageSource<File> sftpMessageSource() {
        SftpInboundFileSynchronizingMessageSource source =
                new SftpInboundFileSynchronizingMessageSource
(sftpInboundFileSynchronizer());
        source.setLocalDirectory(new File("sftp-inbound"));
        source.setAutoCreateLocalDirectory(true);
        source.setLocalFilter(new AcceptOnceFileListFilter<File>());
        source.setMaxFetchSize(1);
        return source;
    }

    @Bean
    @ServiceActivator(inputChannel = "sftpChannel")
    public MessageHandler handler() {
        return new MessageHandler() {
```

```
            @Override
            public void handleMessage(Message<?> message) throws
MessagingException {
                System.out.println(message.getPayload());
            }

        };
    }

}
```

32.6.4. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the inbound adapter with the Java DSL:

```
@SpringBootApplication
public class SftpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(SftpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public IntegrationFlow sftpInboundFlow() {
        return IntegrationFlows
            .from(Sftp.inboundAdapter(this.sftpSessionFactory)
                    .preserveTimestamp(true)
                    .remoteDirectory("foo")
                    .regexFilter(".*\\.txt$")
                    .localFilenameExpression("#this.toUpperCase() + '.a'")
                    .localDirectory(new File("sftp-inbound")),
                e -> e.id("sftpInboundAdapter")
                    .autoStartup(true)
                    .poller(Pollers.fixedDelay(5000)))
            .handle(m -> System.out.println(m.getPayload()))
            .get();
    }
}
```

32.6.5. Dealing With Incomplete Data

See Dealing With Incomplete Data.

The SftpSystemMarkerFilePresentFileListFilter is provided to filter remote files that don't have the corresponding marker file on the remote system. See the Javadoc for configuration information.

32.7. SFTP Streaming Inbound Channel Adapter

Version 4.3 introduced the streaming inbound channel adapter. This adapter produces message with payloads of type InputStream, letting you fetch files without writing to the local file system. Since the session remains open, the consuming application is responsible for closing the session when the file has been consumed. The session is provided in the closeableResource header (IntegrationMessageHeaderAccessor.CLOSEABLE_RESOURCE). Standard framework components, such as the FileSplitter and StreamTransformer, automatically close the session. See File Splitter and Stream Transformer for more information about these components. The following example shows how to configure an SFTP streaming inbound channel adapter:

```
<int-sftp:inbound-streaming-channel-adapter id="ftpInbound"
            channel="ftpChannel"
            session-factory="sessionFactory"
            filename-pattern="*.txt"
            filename-regex=".*\.txt"
            filter="filter"
            filter-expression="@myFilterBean.check(#root)"
            remote-file-separator="/"
            comparator="comparator"
            max-fetch-size="1"
            remote-directory-expression="'foo/bar'">
        <int:poller fixed-rate="1000" />
</int-sftp:inbound-streaming-channel-adapter>
```

You can use only one of filename-pattern, filename-regex, filter, or filter-expression.

Starting with version 5.0, by default, the SftpStreamingMessageSource adapter prevents duplicates for remote files by using SftpPersistentAcceptOnceFileListFilter based on the in-memory SimpleMetadataStore. By default, this filter is also applied together with the filename pattern (or regex) as well. If you need to allow duplicates, you can use the AcceptAllFileListFilter. You can handle any other use cases by using CompositeFileListFilter (or ChainFileListFilter). The Java configuration shown later shows one technique to remove the remote file after processing, avoiding duplicates.

For more information about the SftpPersistentAcceptOnceFileListFilter, and how it is used, see Remote Persistent File List Filters.

You can use the max-fetch-size attribute to limit the number of files fetched on each poll when a fetch is necessary. Set it to 1 and use a persistent filter when running in a clustered environment. See Inbound Channel Adapters: Controlling Remote File Fetching for more information.

The adapter puts the remote directory and the file name in headers (`FileHeaders.REMOTE_DIRECTORY` and `FileHeaders.REMOTE_FILE`, respectively). Starting with version 5.0, the `FileHeaders.REMOTE_FILE_INFO` header provides additional remote file information (in JSON). If you set the `fileInfoJson` property on the `SftpStreamingMessageSource` to `false`, the header contains an `SftpFileInfo` object. You can access the `LsEntry` object provided by the underlying Jsch library by using the `SftpFileInfo.getFileInfo()` method. The `fileInfoJson` property is not available when you use XML configuration, but you can set it by injecting the `SftpStreamingMessageSource` into one of your configuration classes. See also Remote File Information.

Starting with version 5.1, the generic type of the `comparator` is `LsEntry`. Previously, it was `AbstractFileInfo<LsEntry>`. This is because the sort is now performed earlier in the processing, before filtering and applying `maxFetch`.

32.7.1. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the inbound adapter with Java:

```
@SpringBootApplication
public class SftpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(SftpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    @InboundChannelAdapter(channel = "stream")
    public MessageSource<InputStream> ftpMessageSource() {
        SftpStreamingMessageSource messageSource = new SftpStreamingMessageSource
(template());
        messageSource.setRemoteDirectory("sftpSource/");
        messageSource.setFilter(new AcceptAllFileListFilter<>());
        messageSource.setMaxFetchSize(1);
        return messageSource;
    }

    @Bean
    @Transformer(inputChannel = "stream", outputChannel = "data")
    public org.springframework.integration.transformer.Transformer transformer() {
        return new StreamTransformer("UTF-8");
    }

    @Bean
    public SftpRemoteFileTemplate template() {
        return new SftpRemoteFileTemplate(sftpSessionFactory());
    }

    @ServiceActivator(inputChannel = "data", adviceChain = "after")
    @Bean
    public MessageHandler handle() {
        return System.out::println;
    }

    @Bean
    public ExpressionEvaluatingRequestHandlerAdvice after() {
        ExpressionEvaluatingRequestHandlerAdvice advice = new
ExpressionEvaluatingRequestHandlerAdvice();
        advice.setOnSuccessExpression(
                "@template.remove(headers['file_remoteDirectory'] +
headers['file_remoteFile'])");
        advice.setPropagateEvaluationFailures(true);
        return advice;
    }

}
```

Notice that, in this example, the message handler downstream of the transformer has an advice that removes the remote file after processing.

32.8. Inbound Channel Adapters: Polling Multiple Servers and Directories

Starting with version 5.0.7, the `RotatingServerAdvice` is available; when configured as a poller advice, the inbound adapters can poll multiple servers and directories. Configure the advice and add it to the poller's advice chain as normal. A `DelegatingSessionFactory` is used to select the server see Delegating Session Factory for more information. The advice configuration consists of a list of `RotationPolicy.KeyDirectory` objects.

Example

```
@Bean
public RotatingServerAdvice advice() {
    List<RotationPolicy.KeyDirectory> keyDirectories = new ArrayList<>();
    keyDirectories.add(new RotationPolicy.KeyDirectory("one", "foo"));
    keyDirectories.add(new RotationPolicy.KeyDirectory("one", "bar"));
    keyDirectories.add(new RotationPolicy.KeyDirectory("two", "baz"));
    keyDirectories.add(new RotationPolicy.KeyDirectory("two", "qux"));
    keyDirectories.add(new RotationPolicy.KeyDirectory("three", "fiz"));
    keyDirectories.add(new RotationPolicy.KeyDirectory("three", "buz"));
    return new RotatingServerAdvice(delegatingSf(), keyDirectories);
}
```

This advice will poll directory `foo` on server `one` until no new files exist then move to directory `bar` and then directory `baz` on server `two`, etc.

This default behavior can be modified with the `fair` constructor arg:

fair

```
@Bean
public RotatingServerAdvice advice() {
    ...
    return new RotatingServerAdvice(delegatingSf(), keyDirectories, true);
}
```

In this case, the advice will move to the next server/directory regardless of whether the previous poll returned a file.

Alternatively, you can provide your own `RotationPolicy` to reconfigure the message source as needed:

policy

```
public interface RotationPolicy {

    void beforeReceive(MessageSource<?> source);

    void afterReceive(boolean messageReceived, MessageSource<?> source);

}
```

and

custom

```
@Bean
public RotatingServerAdvice advice() {
    return new RotatingServerAdvice(myRotationPolicy());
}
```

The `local-filename-generator-expression` attribute (`localFilenameGeneratorExpression` on the synchronizer) can now contain the `#remoteDirectory` variable. This allows files retrieved from different directories to be downloaded to similar directories locally:

```
@Bean
public IntegrationFlow flow() {
    return IntegrationFlows.from(Sftp.inboundAdapter(sf())
                    .filter(new SftpPersistentAcceptOnceFileListFilter(new
SimpleMetadataStore(), "rotate"))
                    .localDirectory(new File(tmpDir))
                    .localFilenameExpression("#remoteDirectory +
T(java.io.File).separator + #root")
                    .remoteDirectory("."),
                e -> e.poller(Pollers.fixedDelay(1).advice(advice())))
            .channel(MessageChannels.queue("files"))
            .get();
}
```

Do not configure a `TaskExecutor` on the poller when using this advice; see Conditional Pollers for Message Sources for more information.

32.9. Inbound Channel Adapters: Controlling Remote File Fetching

You should consider two properties when configuring inbound channel adapters. `max-messages-per-poll`, as with all pollers, can be used to limit the number of messages emitted on each poll (if more than the configured value are ready). `max-fetch-size` (since version 5.0) can limit the number of

files retrieved from the remote server at a time.

The following scenarios assume the starting state is an empty local directory:

- `max-messages-per-poll=2` and `max-fetch-size=1`: The adapter fetches one file, emits it, fetches the next file, and emit it. Then it sleeps until the next poll.
- `max-messages-per-poll=2` and `max-fetch-size=2`): The adapter fetch both files and then emits each one.
- `max-messages-per-poll=2` and `max-fetch-size=4`: The adapter fetches up to 4 files (if available) and emits the first two (if there are at least two). The next two files will be emitted on the next poll.
- `max-messages-per-poll=2` and `max-fetch-size` not specified: The adapter fetches all remote files and emits the first two (if there are at least two). The subsequent files are emitted on subsequent polls (two at a time). When all are consumed, the remote fetch is attempted again, to pick up any new files.

When you deploy multiple instances of an application, we recommend setting a small `max-fetch-size`, to avoid one instance "grabbing" all the files and starving other instances.

Another use for `max-fetch-size` is when you want to stop fetching remote files but continue to process files that have already been fetched. Setting the `maxFetchSize` property on the `MessageSource` (programmatically, via JMX, or via a control bus) effectively stops the adapter from fetching more files but lets the poller continue to emit messages for files that have previously been fetched. If the poller is active when the property is changed, the change takes effect on the next poll.

Starting with version 5.1, the synchronizer can be provided with a `Comparator<LsEntry>`. This is useful when restricting the number of files fetched with `maxFetchSize`.

32.10. SFTP Outbound Channel Adapter

The SFTP outbound channel adapter is a special `MessageHandler` that connects to the remote directory and initiates a file transfer for every file it receives as the payload of an incoming `Message`. It also supports several representations of the file so that you are not limited to the `File` object. Similar to the FTP outbound adapter, the SFTP outbound channel adapter supports the following payloads:

- `java.io.File`: The actual file object
- `byte[]`: A byte array that represents the file contents
- `java.lang.String`: Text that represents the file contents
- `java.io.InputStream`: a stream of data to transfer to remote file
- `org.springframework.core.io.Resource`: a resource for data to transfer to remote file

The following example shows how to configure an SFTP outbound channel adapter:

```
<int-sftp:outbound-channel-adapter id="sftpOutboundAdapter"
    session-factory="sftpSessionFactory"
    channel="inputChannel"
    charset="UTF-8"
    remote-file-separator="/"
    remote-directory="foo/bar"
    remote-filename-generator-expression="payload.getName() + '-mysuffix'"
    filename-generator="fileNameGenerator"
    use-temporary-filename="true"
    chmod="600"
    mode="REPLACE"/>
```

See the schema for more detail on these attributes.

32.10.1. SpEL and the SFTP Outbound Adapter

As with many other components in Spring Integration, you can use the Spring Expression Language (SpEL) when you configure an SFTP outbound channel adapter by specifying two attributes: `remote-directory-expression` and `remote-filename-generator-expression` (described earlier). The expression evaluation context has the message as its root object, which lets you use expressions that can dynamically compute the file name or the existing directory path based on the data in the message (from either the 'payload' or the 'headers'). In the preceding example, we define the `remote-filename-generator-expression` attribute with an expression value that computes the file name based on its original name while also appending a suffix: '-mysuffix'.

Starting with version 4.1, you can specify the `mode` when you transferring the file. By default, an existing file is overwritten. The modes are defined by the `FileExistsMode` enumeration, which includes the following values:

- `REPLACE` (default)
- `REPLACE_IF_MODIFIED`
- `APPEND`
- `APPEND_NO_FLUSH`
- `IGNORE`
- `FAIL`

With `IGNORE` and `FAIL`, the file is not transferred. `FAIL` causes an exception to be thrown, while `IGNORE` silently ignores the transfer (although a `DEBUG` log entry is produced).

Version 4.3 introduced the `chmod` attribute, which you can use to change the remote file permissions after upload. You can use the conventional Unix octal format (for example, `600` allows read-write for the file owner only). When configuring the adapter using java, you can use `setChmodOctal("600")` or `setChmod(0600)`.

32.10.2. Avoiding Partially Written Files

One of the common problems when dealing with file transfers is the possibility of processing a partial file. A file might appear in the file system before its transfer is actually complete.

To deal with this issue, Spring Integration SFTP adapters use a common algorithm in which files are transferred under a temporary name and than renamed once they are fully transferred.

By default, every file that is in the process of being transferred appear in the file system with an additional suffix, which, by default, is `.writing`. You can change by setting the `temporary-file-suffix` attribute.

However, there may be situations where you do not want to use this technique (for example, if the server does not permit renaming files). For situations like this, you can disable this feature by setting `use-temporary-file-name` to `false` (the default is `true`). When this attribute is `false`, the file is written with its final name, and the consuming application needs some other mechanism to detect that the file is completely uploaded before accessing it.

32.10.3. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the outbound adapter with Java:

```
@SpringBootApplication
@IntegrationComponentScan
public class SftpJavaApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
                    new SpringApplicationBuilder(SftpJavaApplication.class)
                        .web(false)
                        .run(args);
        MyGateway gateway = context.getBean(MyGateway.class);
        gateway.sendToSftp(new File("/foo/bar.txt"));
    }

    @Bean
    public SessionFactory<LsEntry> sftpSessionFactory() {
        DefaultSftpSessionFactory factory = new DefaultSftpSessionFactory(true);
        factory.setHost("localhost");
        factory.setPort(port);
        factory.setUser("foo");
        factory.setPassword("foo");
        factory.setAllowUnknownKeys(true);
        factory.setTestSession(true);
        return new CachingSessionFactory<LsEntry>(factory);
    }

    @Bean
    @ServiceActivator(inputChannel = "toSftpChannel")
    public MessageHandler handler() {
        SftpMessageHandler handler = new SftpMessageHandler(sftpSessionFactory());
        handler.setRemoteDirectoryExpressionString("headers['remote-target-dir']"
);
        handler.setFileNameGenerator(new FileNameGenerator() {

            @Override
            public String generateFileName(Message<?> message) {
                 return "handlerContent.test";
            }

        });
        return handler;
    }

    @MessagingGateway
    public interface MyGateway {

         @Gateway(requestChannel = "toSftpChannel")
         void sendToSftp(File file);

    }
}
```

32.10.4. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the outbound adapter with the Java DSL:

```
@SpringBootApplication
public class SftpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(SftpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public IntegrationFlow sftpOutboundFlow() {
        return IntegrationFlows.from("toSftpChannel")
            .handle(Sftp.outboundAdapter(this.sftpSessionFactory, FileExistsMode
.FAIL)
                           .useTemporaryFileName(false)
                           .remoteDirectory("/foo")
            ).get();
    }

}
```

32.11. SFTP Outbound Gateway

The SFTP outbound gateway provides a limited set of commands that let you interact with a remote SFTP server:

- `ls` (list files)
- `nlst` (list file names)
- `get` (retrieve a file)
- `mget` (retrieve multiple files)
- `rm` (remove file(s))
- `mv` (move and rename file)
- `put` (send a file)
- `mput` (send multiple files)

32.11.1. Using the `ls` Command

`ls` lists remote files and supports the following options:

- `-1`: Retrieve a list of filenames. The default is to retrieve a list of `FileInfo` objects
- `-a`: Include all files (including those starting with '.')
- `-f`: Do not sort the list
- `-dirs`: Include directories (excluded by default)
- `-links`: Include symbolic links (excluded by default)
- `-R`: List the remote directory recursively

In addition, filename filtering is provided in the same manner as the `inbound-channel-adapter`.

The message payload resulting from an `ls` operation is a list of file names or a list of `FileInfo` objects (depending on whether you usr the `-1` switch). These objects provide information such as modified time, permissions, and others.

The remote directory that the `ls` command acted on is provided in the `file_remoteDirectory` header.

When using the recursive option (`-R`), the `fileName` includes any subdirectory elements and represents the relative path to the file (relative to the remote directory). If you use the `-dirs` option, each recursive directory is also returned as an element in the list. In this case, we recommend that you not use the `-1` option, because you would not be able to distinguish files from directories, which you can do when you use `FileInfo` objects.

Using `nlst` Command

Version 5 introduced support for the `nlst` command.

`nlst` lists remote file names and supports only one option:

- `-f`: Do not sort the list

The message payload resulting from an `nlst` operation is a list of file names.

The `file_remoteDirectory` header holds the remote directory on which the `nlst` command acted.

The SFTP protocol does not provide the ability to list names. This command is the equivalent of the `ls` command with the `-1` option and is added here for convenience.

32.11.2. Using the `get` Command

`get` retrieves a remote file and supports the following options:

- `-P`: Preserve the timestamp of the remote file.
- `-stream`: Retrieve the remote file as a stream.
- `-D`: Delete the remote file after successful transfer. The remote file is not deleted if the transfer is ignored, because the `FileExistsMode` is `IGNORE` and the local file already exists.

The `file_remoteDirectory` header holds the remote directory, and the `file_remoteFile` header holds the filename.

The message payload resulting from a `get` operation is a `File` object representing the retrieved file. If you use the `-stream` option, the payload is an `InputStream` rather than a `File`. For text files, a common use case is to combine this operation with a file splitter or a stream transformer. When consuming remote files as streams, you are responsible for closing the `Session` after the stream is consumed. For convenience, the `Session` is provided in the `closeableResource` header, and `IntegrationMessageHeaderAccessor` offers convenience method:

```
Closeable closeable = new IntegrationMessageHeaderAccessor(message)
.getCloseableResource();
if (closeable != null) {
    closeable.close();
}
```

Framework components, such as the File Splitter and Stream Transformer, automatically close the session after the data is transferred.

The following example shows how to consume a file as a stream:

```
<int-sftp:outbound-gateway session-factory="ftpSessionFactory"
                           request-channel="inboundGetStream"
                           command="get"
                           command-options="-stream"
                           expression="payload"
                           remote-directory="ftpTarget"
                           reply-channel="stream" />

<int-file:splitter input-channel="stream" output-channel="lines" />
```

If you consume the input stream in a custom component, you must close the `Session`. You can either do that in your custom code or route a copy of the message to a `service-activator` and use SpEL, as the following example shows:

```
<int:service-activator input-channel="closeSession"
    expression="headers['closeableResource'].close()" />
```

32.11.3. Using the `mget` Command

`mget` retrieves multiple remote files based on a pattern and supports the following options:

- `-P`: Preserve the timestamps of the remote files.
- `-R`: Retrieve the entire directory tree recursively.

- `-x`: Throw an exception if no files match the pattern (otherwise, an empty list is returned).
- `-D`: Delete each remote file after successful transfer. If the transfer is ignored, the remote file is not deleted, because the `FileExistsMode` is `IGNORE` and the local file already exists.

The message payload resulting from an `mget` operation is a `List<File>` object (that is, a `List` of `File` objects, each representing a retrieved file).

Starting with version 5.0, if the `FileExistsMode` is `IGNORE`, the payload of the output message no longer contain files that were not fetched due to the file already existing. Previously, the array contained all files, including those that already existed.

The expression you use determine the remote path should produce a result that ends with `for example myfiles/` fetches the complete tree under `myfiles`.

Starting with version 5.0, you can use a recursive `MGET`, combined with the `FileExistsMode.REPLACE_IF_MODIFIED` mode, to periodically synchronize an entire remote directory tree locally. This mode sets the local file's last modified timestamp to the remote file's timestamp, regardless of the `-P` (preserve timestamp) option.

Notes for when using recursion (`-R`)

The pattern is ignored and `*` is assumed. By default, the entire remote tree is retrieved. However, you can filter files in the tree by providing a `FileListFilter`. You can also filter directories in the tree this way. A `FileListFilter` can be provided by reference or by `filename-pattern` or `filename-regex` attributes. For example, `filename-regex="(subDir|.*1.txt)"` retrieves all files ending with `1.txt` in the remote directory and the subdirectory `subDir`. However, we describe an alternative available after this note.

If you filter a subdirectory, no additional traversal of that subdirectory is performed.

The `-dirs` option is not allowed (the recursive `mget` uses the recursive `ls` to obtain the directory tree and the directories themselves cannot be included in the list).

Typically, you would use the `#remoteDirectory` variable in the `local-directory-expression` so that the remote directory structure is retained locally.

Starting with version 5.0, you can configure the `SftpSimplePatternFileListFilter` and `SftpRegexPatternFileListFilter` to always pass directories by setting the `alwaysAcceptDirectorties` to `true`. Doing so allows recursion for a simple pattern, as the following examples show:

```
<bean id="starDotTxtFilter"
            class=
"org.springframework.integration.sftp.filters.SftpSimplePatternFileListFilter">
    <constructor-arg value="*.txt" />
    <property name="alwaysAcceptDirectories" value="true" />
</bean>

<bean id="dotStarDotTxtFilter"
            class=
"org.springframework.integration.sftp.filters.SftpRegexPatternFileListFilter">
    <constructor-arg value="^.*\.txt$" />
    <property name="alwaysAcceptDirectories" value="true" />
</bean>
```

You can provide one of these filters by using the `filter` property on the gateway.

See also Outbound Gateway Partial Success (`mget` and `mput`).

32.11.4. Using the `put` Command

`put` sends a file to the remote server. The payload of the message can be a `java.io.File`, a `byte[]`, or a `String`. A `remote-filename-generator` (or expression) is used to name the remote file. Other available attributes include `remote-directory`, `temporary-remote-directory` and their `*-expression` equivalents: `use-temporary-file-name` and `auto-create-directory`. See the schema documentation for more information.

The message payload resulting from a `put` operation is a `String` that contains the full path of the file on the server after transfer.

Version 4.3 introduced the `chmod` attribute, which changes the remote file permissions after upload. You can use the conventional Unix octal format (for example, `600` allows read-write for the file owner only). When configuring the adapter using java, you can use `setChmod(0600)`.

32.11.5. Using the `mput` Command

`mput` sends multiple files to the server and supports the following option:

- `-R`: Recursive — send all files (possibly filtered) in the directory and subdirectories

The message payload must be a `java.io.File` (or `String`) that represents a local directory. Since version 5.1, a collection of `File` or `String` is also supported.

The same attributes as the `put` command are supported. In addition, you can filter files in the local directory with one of `mput-pattern`, `mput-regex`, `mput-filter`, or `mput-filter-expression`. The filter works with recursion, as long as the subdirectories themselves pass the filter. Subdirectories that do not pass the filter are not recursed.

The message payload resulting from an `mget` operation is a `List<String>` object (that is, a `List` of

remote file paths resulting from the transfer).

See also Outbound Gateway Partial Success (`mget` and `mput`).

Version 4.3 introduced the `chmod` attribute, which lets you change the remote file permissions after upload. You can use the conventional Unix octal format (for example, `600` allows read-write for the file owner only). When configuring the adapter with Java, you can use `setChmodOctal("600")` or `setChmod(0600)`.

Using the `rm` Command

The `rm` command has no options.

If the remove operation was successful, the resulting message payload is `Boolean.TRUE`. Otherwise, the message payload is `Boolean.FALSE`. The `file_remoteDirectory` header holds the remote directory, and the `file_remoteFile` header holds the file name.

32.11.6. Using the `mv` Command

The `mv` command has no options.

The `expression` attribute defines the "from" path, and the `rename-expression` attribute defines the "to" path. By default, the `rename-expression` is `headers['file_renameTo']`. This expression must not evaluate to null or an empty `String`. If necessary, any remote directories needed are created. The payload of the result message is `Boolean.TRUE`. The the `file_remoteDirectory` header holds the original remote directory, and the `file_remoteFile` header holds the filename. The `file_renameTo` header holds the new path.

32.11.7. Additional Command Information

The `get` and `mget` commands support the `local-filename-generator-expression` attribute. It defines a SpEL expression to generate the names of local files during the transfer. The root object of the evaluation context is the request message. The `remoteFileName` variable is also available. It is particularly useful for `mget` (for example: `local-filename-generator-expression="#remoteFileName.toUpperCase() + headers.foo"`).

The `get` and `mget` commands support the `local-directory-expression` attribute. It defines a SpEL expression to generate the names of local directories during the transfer. The root object of the evaluation context is the request message. The `remoteDirectory` variable is also available. It is particularly useful for mget (for example: `local-directory-expression="'/tmp/local/' + #remoteDirectory.toUpperCase() + headers.myheader"`). This attribute is mutually exclusive with the `local-directory` attribute.

For all commands, the 'expression' property of the gateway holds the path on which the command acts. For the `mget` command, the expression might evaluate to `*`, meaning to retrieve all files, `somedirectory/*`, and other values that end with `*`.

The following example shows a gateway configured for an `ls` command:

```
<int-ftp:outbound-gateway id="gateway1"
        session-factory="ftpSessionFactory"
        request-channel="inbound1"
        command="ls"
        command-options="-1"
        expression="payload"
        reply-channel="toSplitter"/>
```

The payload of the message sent to the `toSplitter` channel is a list of `String` objects, each of which contains the name of a file. If you omitted `command-options="-1"`, the payload would be a list of `FileInfo` objects. You can provide options as a space-delimited list (for example, `command-options="-1 -dirs -links"`).

Starting with version 4.2, the `GET`, `MGET`, `PUT`, and `MPUT` commands support a `FileExistsMode` property (`mode` when using the namespace support). This affects the behavior when the local file exists (`GET` and `MGET`) or the remote file exists (`PUT` and `MPUT`). The supported modes are `REPLACE`, `APPEND`, `FAIL`, and `IGNORE`. For backwards compatibility, the default mode for `PUT` and `MPUT` operations is `REPLACE`. For `GET` and `MGET` operations, the default is `FAIL`.

32.11.8. Configuring with Java Configuration

The following Spring Boot application shows an example of how to configure the outbound gateway with Java:

```
@SpringBootApplication
public class SftpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(SftpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    @ServiceActivator(inputChannel = "sftpChannel")
    public MessageHandler handler() {
        return new SftpOutboundGateway(ftpSessionFactory(), "ls",
"'my_remote_dir/'");
    }

}
```

32.11.9. Configuring with the Java DSL

The following Spring Boot application shows an example of how to configure the outbound gateway with the Java DSL:

```
@SpringBootApplication
public class SftpJavaApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(SftpJavaApplication.class)
            .web(false)
            .run(args);
    }

    @Bean
    public SessionFactory<LsEntry> sftpSessionFactory() {
        DefaultSftpSessionFactory sf = new DefaultSftpSessionFactory();
        sf.setHost("localhost");
        sf.setPort(port);
        sf.setUsername("foo");
        sf.setPassword("foo");
        factory.setTestSession(true);
        return new CachingSessionFactory<LsEntry>(sf);
    }

    @Bean
    public QueueChannelSpec remoteFileOutputChannel() {
        return MessageChannels.queue();
    }

    @Bean
    public IntegrationFlow sftpMGetFlow() {
        return IntegrationFlows.from("sftpMgetInputChannel")
            .handle(Sftp.outboundGateway(sftpSessionFactory(),
                            AbstractRemoteFileOutboundGateway.Command.MGET,
"payload")
                    .options(AbstractRemoteFileOutboundGateway.Option.RECURSIVE)
                    .regexFileNameFilter("(subSftpSource|.*1.txt)")
                    .localDirectoryExpression("'myDir/' + #remoteDirectory")
                    .localFilenameExpression(
"#remoteFileName.replaceFirst('sftpSource', 'localTarget')"))
            .channel("remoteFileOutputChannel")
            .get();
    }

}
```

32.11.10. Outbound Gateway Partial Success (`mget` and `mput`)

When performing operations on multiple files (by using `mget` and `mput`) an exception can occur some time after one or more files have been transferred. In this case (starting with version 4.2), a `PartialSuccessException` is thrown. As well as the usual `MessagingException` properties (`failedMessage` and `cause`), this exception has two additional properties:

- `partialResults`: The successful transfer results.
- `derivedInput`: The list of files generated from the request message (such as local files to transfer for an `mput`).

These attributes let you determine which files were successfully transferred and which were not.

In the case of a recursive `mput`, the `PartialSuccessException` may have nested `PartialSuccessException` instances.

Consider the following directory structure:

```
root/
|- file1.txt
|- subdir/
   | - file2.txt
   | - file3.txt
|- zoo.txt
```

If the exception occurs on `file3.txt`, the `PartialSuccessException` thrown by the gateway has `derivedInput` of `file1.txt`, `subdir`, and `zoo.txt` and `partialResults` of `file1.txt`. Its `cause` is another `PartialSuccessException` with `derivedInput` of `file2.txt` and `file3.txt` and `partialResults` of `file2.txt`.

32.12. SFTP/JSCH Logging

Since we use JSch libraries to provide SFTP support, you may at times require more information from the JSch API itself, especially if something is not working properly (such as authentication exceptions). Unfortunately JSch does not use `commons-logging` but instead relies on custom implementations of their `com.jcraft.jsch.Logger` interface. As of Spring Integration 2.0.1, we have implemented this interface. So now, to enable JSch logging, you can configure your logger the way you usually do. For example, the following example is valid configuration of a logger that uses Log4J:

```
log4j.category.com.jcraft.jsch=DEBUG
```

32.13. MessageSessionCallback

Starting with Spring Integration version 4.2, you can use a `MessageSessionCallback<F, T>` implementation with the `<int-sftp:outbound-gateway/>` (`SftpOutboundGateway`) to perform any operation on the `Session<LsEntry>` with the `requestMessage` context. You can use it for any non-standard or low-level SFTP operation (or several), such as allowing access from an integration flow definition, or functional interface (lambda) implementation injection. The following example uses a lambda:

```
@Bean
@ServiceActivator(inputChannel = "sftpChannel")
public MessageHandler sftpOutboundGateway(SessionFactory<ChannelSftp.LsEntry>
sessionFactory) {
    return new SftpOutboundGateway(sessionFactory,
         (session, requestMessage) -> session.list(requestMessage.getPayload()));
}
```

Another example might be to pre- or post-process the file data being sent or retrieved.

When using XML configuration, the `<int-sftp:outbound-gateway/>` provides a `session-callback` attribute that lets you specify the `MessageSessionCallback` bean name.

The `session-callback` is mutually exclusive with the `command` and `expression` attributes. When configuring with Java, the `SftpOutboundGateway` class offers different constructors.

32.14. Apache Mina SFTP Server Events

The `ApacheMinaSftpEventListener`, added in version 5.2, listens for certain Apache Mina SFTP server events and publishes them as `ApplicationEvent` s which can be received by any `ApplicationListener` bean, `@EventListener` bean method, or Event Inbound Channel Adapter.

Currently supported events are:

- `SessionOpenedEvent` - a client session was opened
- `DirectoryCreatedEvent` - a directory was created
- `FileWrittenEvent` - a file was written to
- `PathMovedEvent` - a file or directory was renamed
- `PathRemovedEvent` - a file or directory was removed
- `SessionClosedEvent` - the client has disconnected

Each of these is a subclass of `ApacheMinaSftpEvent`; you can configure a single listener to receive all of the event types. The `source` property of each event is a `ServerSession`, from which you can obtain information such as the client address; a convenient `getSession()` method is provided on the

abstract event.

To configure the server with the listener (which must be a Spring bean), simply add it to the `SftpSubsystemFactory`:

```
server = SshServer.setUpDefaultServer();
...
SftpSubsystemFactory sftpFactory = new SftpSubsystemFactory();
sftpFactory.addSftpEventListener(apacheMinaSftpEventListenerBean);
...
```

To consume these events using a Spring Integration event adapter:

```
@Bean
public ApplicationEventListeningMessageProducer eventsAdapter() {
    ApplicationEventListeningMessageProducer producer =
        new ApplicationEventListeningMessageProducer();
    producer.setEventTypes(ApacheMinaSftpEvent.class);
    producer.setOutputChannel(eventChannel());
    return producer;
}
```

32.15. Remote File Information

Starting with version 5.2, the `SftpStreamingMessageSource` (SFTP Streaming Inbound Channel Adapter), `SftpInboundFileSynchronizingMessageSource` (SFTP Inbound Channel Adapter) and "read"-commands of the `SftpOutboundGateway` (SFTP Outbound Gateway) provide additional headers in the message to produce with an information about the remote file:

- `FileHeaders.REMOTE_HOST_PORT` - the host:port pair the remote session has been connected to during file transfer operation;
- `FileHeaders.REMOTE_DIRECTORY` - the remote directory the operation has been performed;
- `FileHeaders.REMOTE_FILE` - the remote file name; applicable only for single file operations.

Since the `SftpInboundFileSynchronizingMessageSource` doesn't produce messages against remote files, but using a local copy, the `AbstractInboundFileSynchronizer` stores an information about remote file in the `MetadataStore` (which can be configured externally) in the URI style (`protocol://host:port/remoteDirectory#remoteFileName`) during synchronization operation. This metadata is retrieved by the `SftpInboundFileSynchronizingMessageSource` when local file is polled. When local file is deleted, it is recommended to remove its metadata entry. The `AbstractInboundFileSynchronizer` provides a `removeRemoteFileMetadata()` callback for this purpose. In addition there is a `setMetadataStorePrefix()` to be used in the metadata keys. It is recommended to have this prefix be different from the one used in the `MetadataStore`-based `FileListFilter`

implementations, when the same `MetadataStore` instance is shared between these components, to avoid entry overriding because both filter and `AbstractInboundFileSynchronizer` use the same local file name for the metadata entry key.

Chapter 33. STOMP Support

Spring Integration version 4.2 introduced STOMP (Simple Text Orientated Messaging Protocol) client support. It is based on the architecture, infrastructure, and API from the Spring Framework's messaging module, stomp package. Spring Integration uses many of Spring STOMP components (such as `StompSession` and `StompClientSupport`). For more information, see the Spring Framework STOMP Support chapter in the Spring Framework reference manual.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-stomp</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-stomp:5.3.0.RC1"
```

For server side components you need to add a `org.springframework:spring-websocket` and/or `io.projectreactor.netty:reactor-netty` dependencies.

33.1. Overview

To configure STOMP, you should start with the STOMP client object. The Spring Framework provides the following implementations:

- `WebSocketStompClient`: Built on the Spring WebSocket API with support for standard JSR-356 WebSocket, Jetty 9, and SockJS for HTTP-based WebSocket emulation with SockJS Client.
- `ReactorNettyTcpStompClient`: Built on `ReactorNettyTcpClient` from the `reactor-netty` project.

You can provide any other `StompClientSupport` implementation. See the Javadoc of those classes.

The `StompClientSupport` class is designed as a *factory* to produce a `StompSession` for the provided `StompSessionHandler` and all the remaining work is done through the *callbacks* to that `StompSessionHandler` and `StompSession` abstraction. With the Spring Integration *adapter* abstraction, we need to provide some managed shared object to represent our application as a STOMP client with its unique session. For this purpose, Spring Integration provides the `StompSessionManager` abstraction to manage the *single* `StompSession` between any provided `StompSessionHandler`. This allows the use of *inbound* or *outbound* channel adapters (or both) for the particular STOMP Broker. See `StompSessionManager` (and its implementations) JavaDocs for more information.

33.2. STOMP Inbound Channel Adapter

The `StompInboundChannelAdapter` is a one-stop `MessageProducer` component that subscribes your Spring Integration application to the provided STOMP destinations and receives messages from them (converted from the STOMP frames by using the provided `MessageConverter` on the connected `StompSession`). You can change the destinations (and therefore STOMP subscriptions) at runtime by using appropriate `@ManagedOperation` annotations on the `StompInboundChannelAdapter`.

For more configuration options, see STOMP Namespace Support and the `StompInboundChannelAdapter` Javadoc.

33.3. STOMP Outbound Channel Adapter

The `StompMessageHandler` is the `MessageHandler` for the `<int-stomp:outbound-channel-adapter>` and is used to send the outgoing `Message<?>` instances to the STOMP `destination` (pre-configured or determined at runtime with a SpEL expression) through the `StompSession` (which is provided by the shared `StompSessionManager`).

For more configuration options see STOMP Namespace Support and the `StompMessageHandler` Javadoc.

33.4. STOMP Headers Mapping

The STOMP protocol provides headers as part of its frame. The entire structure of the STOMP frame has the following format:

```
....
COMMAND
header1:value1
header2:value2

Body^@
....
```

Spring Framework provides `StompHeaders` to represent these headers. See the Javadoc for more details. STOMP frames are converted to and from `Message<?>` instances and these headers are mapped to and from `MessageHeaders` instances. Spring Integration provides a default `HeaderMapper` implementation for the STOMP adapters. The implementation is `StompHeaderMapper`. It provides `fromHeaders()` and `toHeaders()` operations for the inbound and outbound adapters, respectively.

As with many other Spring Integration modules, the `IntegrationStompHeaders` class has been introduced to map standard STOMP headers to `MessageHeaders`, with `stomp_` as the header name prefix. In addition, all `MessageHeaders` instances with that prefix are mapped to the `StompHeaders` when sending to a destination.

For more information, see the Javadoc for those classes and the `mapped-headers` attribute description

in the STOMP Namespace Support.

33.5. STOMP Integration Events

Many STOMP operations are asynchronous, including error handling. For example, STOMP has a `RECEIPT` server frame that it returns when a client frame has requested one by adding the `RECEIPT` header. To provide access to these asynchronous events, Spring Integration emits `StompIntegrationEvent` instances, which you can obtain by implementing an `ApplicationListener` or by using an `<int-event:inbound-channel-adapter>` (see Receiving Spring Application Events).

Specifically, a `StompExceptionEvent` is emitted from the `AbstractStompSessionManager` when a `stompSessionListenableFuture` receives `onFailure()` due to failure to connect to STOMP broker. Another example is the `StompMessageHandler`. It processes `ERROR` STOMP frames, which are server responses to improper (unaccepted) messages sent by this `StompMessageHandler`.

The `StompMessageHandler` emits `StompReceiptEvent` as a part of `StompSession.Receiptable` callbacks in the asynchronous answers for the messages sent to the `StompSession`. The `StompReceiptEvent` can be positive or negative, depending on whether or not the `RECEIPT` frame was received from the server within the `receiptTimeLimit` period, which you can configure on the `StompClientSupport` instance. It defaults to `15 * 1000` (in milliseconds, so 15 seconds).

The `StompSession.Receiptable` callbacks are added only if the `RECEIPT` STOMP header of the message to send is not `null`. You can enable automatic `RECEIPT` header generation on the `StompSession` through its `autoReceipt` option and on the `StompSessionManager` respectively.

See STOMP Adapters Java Configuration for more information how to configure Spring Integration to accept those `ApplicationEvent` instances.

33.6. STOMP Adapters Java Configuration

The following example shows a comprehensive Java configuration for STOMP adapters:

```
@Configuration
@EnableIntegration
public class StompConfiguration {

    @Bean
    public ReactorNettyTcpStompClient stompClient() {
        ReactorNettyTcpStompClient stompClient = new ReactorNettyTcpStompClient(
"127.0.0.1", 61613);
        stompClient.setMessageConverter(new PassThruMessageConverter());
        ThreadPoolTaskScheduler taskScheduler = new ThreadPoolTaskScheduler();
        taskScheduler.afterPropertiesSet();
        stompClient.setTaskScheduler(taskScheduler);
        stompClient.setReceiptTimeLimit(5000);
        return stompClient;
    }

    @Bean
    public StompSessionManager stompSessionManager() {
        ReactorNettyTcpStompSessionManager stompSessionManager = new
ReactorNettyTcpStompSessionManager(stompClient());
        stompSessionManager.setAutoReceipt(true);
        return stompSessionManager;
    }

    @Bean
    public PollableChannel stompInputChannel() {
        return new QueueChannel();
    }

    @Bean
    public StompInboundChannelAdapter stompInboundChannelAdapter() {
        StompInboundChannelAdapter adapter =
                new StompInboundChannelAdapter(stompSessionManager(),
"/topic/myTopic");
        adapter.setOutputChannel(stompInputChannel());
        return adapter;
    }

    @Bean
    @ServiceActivator(inputChannel = "stompOutputChannel")
    public MessageHandler stompMessageHandler() {
        StompMessageHandler handler = new StompMessageHandler(stompSessionManager
());
        handler.setDestination("/topic/myTopic");
        return handler;
    }

    @Bean
    public PollableChannel stompEvents() {
```

```
        return new QueueChannel();
    }

    @Bean
    public ApplicationListener<ApplicationEvent> stompEventListener() {
        ApplicationEventListeningMessageProducer producer = new
ApplicationEventListeningMessageProducer();
        producer.setEventTypes(StompIntegrationEvent.class);
        producer.setOutputChannel(stompEvents());
        return producer;
    }

}
```

33.7. STOMP Namespace Support

The Spring Integration STOMP namespace implements the inbound and outbound channel adapter components. To include it in your configuration, provide the following namespace declaration in your application context configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-stomp="http://www.springframework.org/schema/integration/stomp"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    https://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/stomp
    https://www.springframework.org/schema/integration/stomp/spring-integration-
stomp.xsd">
    ...
</beans>
```

33.7.1. Understanding the `<int-stomp:outbound-channel-adapter>` Element

The following listing shows the available attributes for the STOMP outbound channel adapter:

```
<int-stomp:outbound-channel-adapter
                            id=""                       ①
                            channel=""                  ②
                            stomp-session-manager=""    ③
                            header-mapper=""            ④
                            mapped-headers=""           ⑤
                            destination=""              ⑥
                            destination-expression=""   ⑦
                            auto-startup=""             ⑧
                            phase=""/>                  ⑨
```

① The component bean name. The `MessageHandler` is registered with a bean alias of `id` plus `.handler`. If you do not set the `channel` attribute, a `DirectChannel` is created and registered in the application context with the value of this `id` attribute as the bean name. In this case, the endpoint is registered with a bean name `id` plus `.adapter`.

② Identifies the channel attached to this adapter if `id` is present. See `id`. Optional.

③ Reference to a `StompSessionManager` bean, which encapsulates the low-level connection and `StompSession` handling operations. Required.

④ Reference to a bean that implements `HeaderMapper<StompHeaders>`, which maps Spring Integration `MessageHeaders` to and from STOMP frame headers. It is mutually exclusive with `mapped-headers`. It defaults to `StompHeaderMapper`.

⑤ Comma-separated list of names of STOMP Headers to be mapped to the STOMP frame headers. It can be provided only if the `header-mapper` reference is not set. The values in this list can also be simple patterns to be matched against the header names (such as `myheader*` or `*myheader`). A special token (`STOMP_OUTBOUND_HEADERS`) represents all the standard STOMP headers (content-length, receipt, heart-beat, and so on). They are included by default. If you want to add your own headers and want the standard headers to also be mapped, you must include this token or provide your own `HeaderMapper` implementation by using `header-mapper`.

⑥ Name of the destination to which STOMP Messages are sent. It is mutually exclusive with the `destination-expression`.

⑦ A SpEL expression to be evaluated at runtime against each Spring Integration `Message` as the root object. It is mutually exclusive with the `destination`.

⑧ Boolean value indicating whether this endpoint should start automatically. It defaults to `true`.

⑨ The lifecycle phase within which this endpoint should start and stop. The lower the value, the earlier this endpoint starts and the later it stops. The default is `Integer.MIN_VALUE`. Values can be negative. See `SmartLifeCycle`.

33.7.2. Understanding the `<int-stomp:inbound-channel-adapter>` Element

The following listing shows the available attributes for the STOMP inbound channel adapter:

```
<int-stomp:inbound-channel-adapter
                            id=""                       ①
                            channel=""                  ②
                            error-channel=""            ③
                            stomp-session-manager=""    ④
                            header-mapper=""            ⑤
                            mapped-headers=""           ⑥
                            destinations=""             ⑦
                            send-timeout=""             ⑧
                            payload-type=""             ⑨
                            auto-startup=""             ⑩
                            phase=""/>                  ⑪
```

① The component bean name. If you do not set the `channel` attribute, a `DirectChannel` is created and registered in the application context with the value of this `id` attribute as the bean name. In this case, the endpoint is registered with the bean name `id` plus `.adapter`.

② Identifies the channel attached to this adapter.

③ The `MessageChannel` bean reference to which `ErrorMessage` instances should be sent.

④ See the same option on the `<int-stomp:outbound-channel-adapter>`.

⑤ Comma-separated list of names of STOMP Headers to be mapped from the STOMP frame headers. You can only provide this if the `header-mapper` reference is not set. The values in this list can also be simple patterns to be matched against the header names (for example, `myheader*` or `*myheader`). A special token (`STOMP_INBOUND_HEADERS`) represents all the standard STOMP headers (content-length, receipt, heart-beat, and so on). They are included by default. If you want to add your own headers and want the standard headers to also be mapped, you must also include this token or provide your own `HeaderMapper` implementation using `header-mapper`.

⑥ See the same option on the `<int-stomp:outbound-channel-adapter>`.

⑦ Comma-separated list of STOMP destination names to subscribe. The list of destinations (and therefore subscriptions) can be modified at runtime through the `addDestination()` and `removeDestination()` `@ManagedOperation` annotations.

⑧ Maximum amount of time (in milliseconds) to wait when sending a message to the channel if the channel can block. For example, a `QueueChannel` can block until space is available if its maximum capacity has been reached.

⑨ Fully qualified name of the Java type for the target `payload` to convert from the incoming STOMP frame. It defaults to `String.class`.

⑩ See the same option on the `<int-stomp:outbound-channel-adapter>`.

⑪ See the same option on the `<int-stomp:outbound-channel-adapter>`.

Chapter 34. Stream Support

In many cases, application data is obtained from a stream. It is not recommended to send a reference to a stream as a message payload to a consumer. Instead, messages are created from data that is read from an input stream, and message payloads are written to an output stream one by one.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-stream</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-stream:5.3.0.RC1"
```

34.1. Reading from Streams

Spring Integration provides two adapters for streams. Both `ByteStreamReadingMessageSource` and `CharacterStreamReadingMessageSource` implement `MessageSource`. By configuring one of these within a channel-adapter element, the polling period can be configured and the message bus can automatically detect and schedule them. The byte stream version requires an `InputStream`, and the character stream version requires a `Reader` as the single constructor argument. The `ByteStreamReadingMessageSource` also accepts the 'bytesPerMessage' property to determine how many bytes it tries to read into each `Message`. The default value is 1024. The following example creates an input stream that creates messages that each contain 2048 bytes:

```
<bean class="
org.springframework.integration.stream.ByteStreamReadingMessageSource">
  <constructor-arg ref="someInputStream"/>
  <property name="bytesPerMessage" value="2048"/>
</bean>

<bean class=
"org.springframework.integration.stream.CharacterStreamReadingMessageSource">
  <constructor-arg ref="someReader"/>
</bean>
```

The `CharacterStreamReadingMessageSource` wraps the reader in a `BufferedReader` (if it is not one already). You can set the buffer size used by the buffered reader in the second constructor argument. Starting with version 5.0, a third constructor argument (`blockToDetectEOF`) controls the behavior of the `CharacterStreamReadingMessageSource`. When `false` (the default), the `receive()` method checks whether the reader is `ready()` and returns null if not. EOF (end of file) is not detected in this case. When `true`, the `receive()` method blocks until data is available or EOF is detected on the underlying stream. When EOF is detected, a `StreamClosedEvent` (application event) is published. You can consume this event with a bean that implements `ApplicationListener<StreamClosedEvent>`.

To facilitate EOF detection, the poller thread blocks in the `receive()` method until either data arrives or EOF is detected.

The poller continues to publish an event on each poll once EOF has been detected. The application listener can stop the adapter to prevent this. The event is published on the poller thread. Stopping the adapter causes the thread to be interrupted. If you intend to perform some interruptible task after stopping the adapter, you must either perform the `stop()` on a different thread or use a different thread for those downstream activities. Note that sending to a `QueueChannel` is interruptible, so, if you wish to send a message from the listener, do it before stopping the adapter.

This facilitates “piping” or redirecting data to `stdin`, as the following two examples shows:

```
cat myfile.txt | java -jar my.jar
```

```
java -jar my.jar < foo.txt
```

This approach lets the application terminate when the pipe is closed.

Four convenient factory methods are available:

```
public static final CharacterStreamReadingMessageSource stdin() { ... }

public static final CharacterStreamReadingMessageSource stdin(String charsetName)
{ ... }

public static final CharacterStreamReadingMessageSource stdinPipe() { ... }

public static final CharacterStreamReadingMessageSource stdinPipe(String
charsetName) { ... }
```

34.2. Writing to Streams

For target streams, you can use either of two implementations: `ByteStreamWritingMessageHandler` or `CharacterStreamWritingMessageHandler`. Each requires a single constructor argument (`OutputStream` for byte streams or `Writer` for character streams), and each provides a second constructor that adds the optional 'bufferSize'. Since both of these ultimately implement the `MessageHandler` interface, you can reference them from a `channel-adapter` configuration, as described in Channel Adapter.

```
<bean class=
"org.springframework.integration.stream.ByteStreamWritingMessageHandler">
  <constructor-arg ref="someOutputStream"/>
  <constructor-arg value="1024"/>
</bean>

<bean class=
"org.springframework.integration.stream.CharacterStreamWritingMessageHandler">
  <constructor-arg ref="someWriter"/>
</bean>
```

34.3. Stream Namespace Support

Spring Integration defines a namespace to reduce the configuration needed for stream-related channel adapters. The following schema locations are needed to use it:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:int-stream=
"http://www.springframework.org/schema/integration/stream"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
      https://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/integration/stream
      https://www.springframework.org/schema/integration/stream/spring-
integration-stream.xsd">
```

The following code snippet shows the different configuration options that are supported to configure the inbound channel adapter:

```
<int-stream:stdin-channel-adapter id="adapterWithDefaultCharset"/>

<int-stream:stdin-channel-adapter id="adapterWithProvidedCharset" charset="UTF-8
"/>
```

Starting with version 5.0, you can set the `detect-eof` attribute, which sets the `blockToDetectEOF` property. See Reading from Streams for more information.

To configure the outbound channel adapter, you can use the namespace support as well. The following example shows the different configuration for an outbound channel adapters:

```
<int-stream:stdout-channel-adapter id="stdoutAdapterWithDefaultCharset"
    channel="testChannel"/>

<int-stream:stdout-channel-adapter id="stdoutAdapterWithProvidedCharset" charset=
"UTF-8"
    channel="testChannel"/>

<int-stream:stderr-channel-adapter id="stderrAdapter" channel="testChannel"/>

<int-stream:stdout-channel-adapter id="newlineAdapter" append-newline="true"
    channel="testChannel"/>
```

Chapter 35. Syslog Support

Spring Integration 2.2 introduced the syslog transformer: `SyslogToMapTransformer`.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-syslog</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-syslog:5.3.0.RC1"
```

This transformer, together with a `UDP` or `TCP` inbound adapter, could be used to receive and analyze syslog records from other hosts. The transformer creates a message payload that contains a map of the elements from the syslog message.

Spring Integration 3.0 introduced convenient namespace support for configuring a syslog inbound adapter in a single element.

Starting with version 4.1.1, the framework now supports the extended syslog format, as specified in RFC 5424>. In addition, when using TCP and RFC5424, both `octet counting` and `non-transparent framing` described in RFC 6587 are supported.

35.1. Syslog Inbound Channel Adapter

This element encompasses a `UDP` or `TCP` inbound channel adapter and a `MessageConverter` to convert the syslog message to a Spring Integration message. The `DefaultMessageConverter` delegates to the `SyslogToMapTransformer`, creating a message with its payload being the `Map` of syslog fields. In addition, all fields except the message are also made available as headers in the message and are prefixed with `syslog_`. In this mode, only RFC 3164 (BSD) syslogs are supported.

Since version 4.1, the `DefaultMessageConverter` has a property called `asMap` (the default is `true`). When it is `false`, the converter leaves the message payload as the original complete syslog message (in a `byte[]`) while still setting the headers.

Since version 4.1.1, RFC 5424 is also supported, by using the `RFC5424MessageConverter`. In this case, the fields are not copied as headers, unless `asMap` is set to `false`, in which case the original message is the payload and the decoded fields are headers.

To use RFC 5424 with a TCP transport, you must provide additional configuration to enable the different framing techniques described in RFC 6587. The adapter needs a TCP connection factory that is configured with a `RFC6587SyslogDeserializer`. By default, this deserializer handles `octet counting` and `non-transparent framing` by using a linefeed (LF) to delimit syslog messages. It uses a `ByteArrayLfSerializer` when `octet counting` is not detected. To use different `non-transparent` framing, you can provide it with some other deserializer. While the deserializer can support both `octet counting` and `non-transparent framing`, only one form of the latter is supported. If `asMap` is `false` on the converter, you must set the `retainOriginal` constructor argument in the `RFC6587SyslogDeserializer`.

35.1.1. Example Configuration

The following example defines a `UDP` adapter that sends messages to the `syslogIn` channel (the adapter bean name is `syslogIn.adapter`):

```
<int-syslog:inbound-channel-adapter id="syslogIn" port="1514" />
```

The adapter listens on port `1514`.

The following example defines a `UDP` adapter that sends messages to the `fromSyslog` channel (the adapter bean name is `syslogIn`):

```
<int-syslog:inbound-channel-adapter id="syslogIn"
    channel="fromSyslog" port="1514" />
```

The adapter listens on port `1514`.

The following example defines a `TCP` adapter that sends messages to channel `syslogIn` (the adapter bean name is `syslogIn.adapter`):

```
<int-syslog:inbound-channel-adapter id="bar" protocol="tcp" port="1514" />
```

The adapter listens on port `1514`.

Note the addition of the `protocol` attribute. This attribute can contain `udp` or `tcp`. It defaults to `udp`.

The following example shows a `UDP` adapter that sends messages to channel `fromSyslog`:

```
<int-syslog:inbound-channel-adapter id="udpSyslog"
    channel="fromSyslog"
    auto-startup="false"
    phase="10000"
    converter="converter"
    send-timeout="1000"
    error-channel="errors">
        <int-syslog:udp-attributes port="1514" lookup-host="false" />
</int-syslog:inbound-channel-adapter>
```

The preceding example also shows two `SmartLifecycle` attributes: `auto-startup` and `phase`. It has a reference to a custom `org.springframework.integration.syslog.MessageConverter` with an ID of `converter` and an `error-channel`. Also notice the `udp-attributes` child element. You can set various UDP attributes here, as defined in .UDP Inbound Channel Adapter Attributes.

When you use the `udp-attributes` element, you must provide the `port` attribute there rather than on the `inbound-channel-adapter` element itself.

The following example shows a `TCP` adapter that sends messages to channel `fromSyslog`:

```
<int-syslog:inbound-channel-adapter id="TcpSyslog"
    protocol="tcp"
    channel="fromSyslog"
    connection-factory="cf" />

<int-ip:tcp-connection-factory id="cf" type="server" port="1514" />
```

It also shows how to reference an externally defined connection factory, which can be used for advanced configuration (socket keep-alive and other uses). For more information, see TCP Connection Factories.

The externally configured `connection-factory` must be of type `server`, and the port is defined there rather than on the `inbound-channel-adapter` element itself.

The following example shows a `TCP` adapter that sends messages to channel `fromSyslog`:

```
<int-syslog:inbound-channel-adapter id="rfc5424Tcp"
    protocol="tcp"
    channel="fromSyslog"
    connection-factory="cf"
    converter="rfc5424" />

<int-ip:tcp-connection-factory id="cf"
    using-nio="true"
    type="server"
    port="1514"
    deserializer="rfc6587" />

<bean id="rfc5424" class=
"org.springframework.integration.syslog.RFC5424MessageConverter" />

<bean id="rfc6587" class=
"org.springframework.integration.syslog.inbound.RFC6587SyslogDeserializer" />
```

The preceding example is configured to use the `RFC 5424` converter and is configured with a reference to an externally defined connection factory with the `RFC 6587` deserializer (required for RFC 5424).

Chapter 36. TCP and UDP Support

Spring Integration provides channel adapters for receiving and sending messages over internet protocols. Both UDP (User Datagram Protocol) and TCP (Transmission Control Protocol) adapters are provided. Each adapter provides for one-way communication over the underlying protocol. In addition, Spring Integration provides simple inbound and outbound TCP gateways. These are used when two-way communication is needed.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-ip</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-ip:5.3.0.RC1"
```

36.1. Introduction

Two flavors each of UDP inbound and outbound channel adapters are provided:

- `UnicastSendingMessageHandler` sends a datagram packet to a single destination.
- `UnicastReceivingChannelAdapter` receives incoming datagram packets.
- `MulticastSendingMessageHandler` sends (broadcasts) datagram packets to a multicast address.
- `MulticastReceivingChannelAdapter` receives incoming datagram packets by joining to a multicast address.

TCP inbound and outbound channel adapters are provided:

- `TcpSendingMessageHandler` sends messages over TCP.
- `TcpReceivingChannelAdapter` receives messages over TCP.

An inbound TCP gateway is provided. It allows for simple request-response processing. While the gateway can support any number of connections, each connection can only be processed serially. The thread that reads from the socket waits for, and sends, the response before reading again. If the connection factory is configured for single use connections, the connection is closed after the socket times out.

An outbound TCP gateway is provided. It allows for simple request-response processing. If the associated connection factory is configured for single-use connections, a new connection is

immediately created for each new request. Otherwise, if the connection is in use, the calling thread blocks on the connection until either a response is received or a timeout or I/O error occurs.

The TCP and UDP inbound channel adapters and the TCP inbound gateway support the `error-channel` attribute. This provides the same basic functionality as described in Enter the `GatewayProxyFactoryBean`.

36.2. UDP Adapters

This section describes how to configure and use the UDP adapters.

36.2.1. Outbound UDP Adapters (XML Configuration)

The following example configures a UDP outbound channel adapter:

```
<int-ip:udp-outbound-channel-adapter id="udpOut"
    host="somehost"
    port="11111"
    multicast="false"
    channel="exampleChannel"/>
```

When setting `multicast` to `true`, you should also provide the multicast address in the host attribute.

UDP is an efficient but unreliable protocol. Spring Integration adds two attributes to improve reliability: `check-length` and `acknowledge`. When `check-length` is set to `true`, the adapter precedes the message data with a length field (four bytes in network byte order). This enables the receiving side to verify the length of the packet received. If a receiving system uses a buffer that is too short to contain the packet, the packet can be truncated. The `length` header provides a mechanism to detect this.

Starting with version 4.3, you can set the `port` to `0`, in which case the operating system chooses the port. The chosen port can be discovered by invoking `getPort()` after the adapter is started and `isListening()` returns `true`.

The preceding example shows an outbound channel adapter that adds length checking to the datagram packets:

```
<int-ip:udp-outbound-channel-adapter id="udpOut"
    host="somehost"
    port="11111"
    multicast="false"
    check-length="true"
    channel="exampleChannel"/>
```

The recipient of the packet must also be configured to expect a length to precede the actual data. For a Spring Integration UDP inbound channel adapter, set its `check-length` attribute.

The second reliability improvement allows an application-level acknowledgment protocol to be used. The receiver must send an acknowledgment to the sender within a specified time.

The following example shows an outbound channel adapter that adds length checking to the datagram packets and waits for an acknowledgment:

```
<int-ip:udp-outbound-channel-adapter id="udpOut"
    host="somehost"
    port="11111"
    multicast="false"
    check-length="true"
    acknowledge="true"
    ack-host="thishost"
    ack-port="22222"
    ack-timeout="10000"
    channel="exampleChannel"/>
```

Setting `acknowledge` to `true` implies that the recipient of the packet can interpret the header added to the packet containing acknowledgment data (host and port). Most likely, the recipient is a Spring Integration inbound channel adapter.

When multicast is true, an additional attribute (`min-acks-for-success`) specifies how many acknowledgments must be received within the `ack-timeout`.

Starting with version 4.3, you can set the `ackPort` to `0`, in which case the operating system chooses the port.

36.2.2. Outbound UDP Adapters (Java Configuration)

The following example shows how to configure an outbound UDP adapter with Java:

```
@Bean
@ServiceActivator(inputChannel = "udpOut")
public UnicastSendingMessageHandler handler() {
    return new UnicastSendingMessageHandler("localhost", 11111);
}
```

(or `MulticastSendingChannelAdapter` for multicast).

36.2.3. Outbound UDP Adapters (Java DSL Configuration)

The following example shows how to configure an outbound UDP adapter with the Java DSL:

```
@Bean
public IntegrationFlow udpOutFlow() {
    return IntegrationFlows.from("udpOut")
            .handle(Udp.outboundAdapter("localhost", 1234))
            .get();
}
```

36.2.4. Inbound UDP Adapters (XML Configuration)

The following example shows how to configure a basic unicast inbound udp channel adapter.

```
<int-ip:udp-inbound-channel-adapter id="udpReceiver"
    channel="udpOutChannel"
    port="11111"
    receive-buffer-size="500"
    multicast="false"
    check-length="true"/>
```

The following example shows how to configure a basic multicast inbound udp channel adapter:

```
<int-ip:udp-inbound-channel-adapter id="udpReceiver"
    channel="udpOutChannel"
    port="11111"
    receive-buffer-size="500"
    multicast="true"
    multicast-address="225.6.7.8"
    check-length="true"/>
```

By default, reverse DNS lookups are done on inbound packets to convert IP addresses to host names for use in message headers. In environments where DNS is not configured, this can cause delays. You can override this default behavior by setting the `lookup-host` attribute to `false`.

36.2.5. Inbound UDP Adapters (Java Configuration)

The following example shows how to configure an inbound UDP adapter with Java:

```
@Bean
public UnicastReceivingChannelAdapter udpIn() {
    UnicastReceivingChannelAdapter adapter = new UnicastReceivingChannelAdapter(11111
);
    adapter.setOutputChannelName("udpChannel");
    return adapter;
}
```

The following example shows how to configure an inbound UDP adapter with the Java DSL:

36.2.6. Inbound UDP Adapters (Java DSL Configuration)

```
@Bean
public IntegrationFlow udpIn() {
    return IntegrationFlows.from(Udp.inboundAdapter(11111))
            .channel("udpChannel")
            .get();
}
```

36.2.7. Server Listening Events

Starting with version 5.0.2, a `UdpServerListeningEvent` is emitted when an inbound adapter is started and has begun listening. This is useful when the adapter is configured to listen on port 0, meaning that the operating system chooses the port. It can also be used instead of polling `isListening()`, if you need to wait before starting some other process that will connect to the socket.

36.2.8. Advanced Outbound Configuration

The `<int-ip:udp-outbound-channel-adapter>` (`UnicastSendingMessageHandler`) has `destination-expression` and `socket-expression` options.

You can use the `destination-expression` as a runtime alternative to the hardcoded `host-port` pair to determine the destination address for the outgoing datagram packet against a `requestMessage` (with the root object for the evaluation context). The expression must evaluate to an `URI`, a `String` in the URI style (see RFC-2396), or a `SocketAddress`. You can also use the inbound `IpHeaders.PACKET_ADDRESS` header for this expression. In the framework, the `DatagramPacketMessageMapper` populates this header when we receive datagrams in the `UnicastReceivingChannelAdapter` and convert them to messages. The header value is exactly the result of `DatagramPacket.getSocketAddress()` of the incoming datagram.

With the socket-expression, the outbound channel adapter can use (for example) an inbound channel adapter socket to send datagrams through the same port which they were received. It is useful in a scenario where our application works as a UDP server and clients operate behind network address translation (NAT). This expression must evaluate to a DatagramSocket. The requestMessage is used as the root object for the evaluation context. You cannot use the socket-expression parameter with the multicast and acknowledge parameters. The following example shows how to configure a UDP inbound channel adapter with a transformer that converts to upper case and uses a socket:

```
<int-ip:udp-inbound-channel-adapter id="inbound" port="0" channel="in" />

<int:channel id="in" />

<int:transformer expression="new String(payload).toUpperCase()"
                        input-channel="in" output-channel="out"/>

<int:channel id="out" />

<int-ip:udp-outbound-channel-adapter id="outbound"
                         socket-expression="@inbound.socket"
                         destination-expression="headers['ip_packetAddress']"
                         channel="out" />
```

The following example shows the equivalent configuration with the Java DSL:

```
@Bean
public IntegrationFlow udpEchoUpcaseServer() {
    return IntegrationFlows.from(Udp.inboundAdapter(11111).id("udpIn"))
            .<byte[], String>transform(p -> new String(p).toUpperCase())
            .handle(Udp.outboundAdapter("headers['ip_packetAddress']")
                    .socketExpression("@udpIn.socket"))
            .get();
}
```

36.3. TCP Connection Factories

36.3.1. Overview

For TCP, the configuration of the underlying connection is provided by using a connection factory. Two types of connection factory are provided: a client connection factory and a server connection factory. Client connection factories establish outgoing connections. Server connection factories listen for incoming connections.

An outbound channel adapter uses a client connection factory, but you can also provide a reference

to a client connection factory to an inbound channel adapter. That adapter receives any incoming messages that are received on connections created by the outbound adapter.

An inbound channel adapter or gateway uses a server connection factory. (In fact, the connection factory cannot function without one). You can also provide a reference to a server connection factory to an outbound adapter. You can then use that adapter to send replies to incoming messages on the same connection.

Reply messages are routed to the connection only if the reply contains the `ip_connectionId` header that was inserted into the original message by the connection factory.

This is the extent of message correlation performed when sharing connection factories between inbound and outbound adapters. Such sharing allows for asynchronous two-way communication over TCP. By default, only payload information is transferred using TCP. Therefore, any message correlation must be performed by downstream components such as aggregators or other endpoints. Support for transferring selected headers was introduced in version 3.0. For more information, see TCP Message Correlation.

You may give a reference to a connection factory to a maximum of one adapter of each type.

Spring Integration provides connection factories that use `java.net.Socket` and `java.nio.channel.SocketChannel`.

The following example shows a simple server connection factory that uses `java.net.Socket` connections:

```
<int-ip:tcp-connection-factory id="server"
    type="server"
    port="1234"/>
```

The following example shows a simple server connection factory that uses `java.nio.channel.SocketChannel` connections:

```
<int-ip:tcp-connection-factory id="server"
    type="server"
    port="1234"
    using-nio="true"/>
```

Starting with Spring Integration version 4.2, if the server is configured to listen on a random port (by setting the port to `0`), you can get the actual port chosen by the OS by using `getPort()`. Also, `getServerSocketAddress()` lets you get the complete `SocketAddress`. See the Javadoc for the `TcpServerConnectionFactory` interface for more information.

```
<int-ip:tcp-connection-factory id="client"
    type="client"
    host="localhost"
    port="1234"
    single-use="true"
    so-timeout="10000"/>
```

The following example shows a client connection factory that uses `java.net.Socket` connections and creates a new connection for each message:

```
<int-ip:tcp-connection-factory id="client"
    type="client"
    host="localhost"
    port="1234"
    single-use="true"
    so-timeout="10000"
    using-nio=true/>
```

Starting with version 5.2, the client connection factories support the property `connectTimeout`, specified in seconds, which defaults to 60.

36.3.2. Message Demarcation (Serializers and Deserializers)

TCP is a streaming protocol. This means that some structure has to be provided to data transported over TCP so that the receiver can demarcate the data into discrete messages. Connection factories are configured to use serializers and deserializers to convert between the message payload and the bits that are sent over TCP. This is accomplished by providing a deserializer and a serializer for inbound and outbound messages, respectively. Spring Integration provides a number of standard serializers and deserializers.

`ByteArrayCrlfSerializer`* converts a byte array to a stream of bytes followed by carriage return and linefeed characters (`\r\n`). This is the default serializer (and deserializer) and can be used (for example) with telnet as a client.

The `ByteArraySingleTerminatorSerializer`* converts a byte array to a stream of bytes followed by a single termination character (the default is `0x00`).

The `ByteArrayLfSerializer`* converts a byte array to a stream of bytes followed by a single linefeed

character (0x0a).

The ByteArrayStxEtxSerializer* converts a byte array to a stream of bytes preceded by an STX (0x02) and followed by an ETX (0x03).

The ByteArrayLengthHeaderSerializer converts a byte array to a stream of bytes preceded by a binary length in network byte order (big endian). This an efficient deserializer because it does not have to parse every byte to look for a termination character sequence. It can also be used for payloads that contain binary data. The preceding serializers support only text in the payload. The default size of the length header is four bytes (an Integer), allowing for messages up to (2^31 - 1) bytes. However, the length header can be a single byte (unsigned) for messages up to 255 bytes, or an unsigned short (2 bytes) for messages up to (2^16 - 1) bytes. If you need any other format for the header, you can subclass ByteArrayLengthHeaderSerializer and provide implementations for the readHeader and writeHeader methods. The absolute maximum data size is (2^31 - 1) bytes. Starting with version 5.2, the header value can include the length of the header in addition to the payload. Set the inclusive property to enable that mechanism (it must be set to the same for producers and consumers).

The ByteArrayRawSerializer*, converts a byte array to a stream of bytes and adds no additional message demarcation data. With this serializer (and deserializer), the end of a message is indicated by the client closing the socket in an orderly fashion. When using this serializer, message reception hangs until the client closes the socket or a timeout occurs. A timeout does not result in a message. When this serializer is being used and the client is a Spring Integration application, the client must use a connection factory that is configured with single-use="true". Doing so causes the adapter to close the socket after sending the message. The serializer does not, by itself, close the connection. You should use this serializer only with the connection factories used by channel adapters (not gateways), and the connection factories should be used by either an inbound or outbound adapter but not both. See also ByteArrayElasticRawDeserializer, later in this section. However, since version 5.2, the outbound gateway has a new property closeStreamAfterSend; this allows the use of raw serializers/deserializers because the EOF is signaled to the server, while leaving the connection open to receive the reply.

Before version 4.2.2, when using non-blocking I/O (NIO), this serializer treated a timeout (during read) as an end of file, and the data read so far was emitted as a message. This is unreliable and should not be used to delimit messages. It now treats such conditions as an exception. In the unlikely event that you use it this way, you can restore the previous behavior by setting the treatTimeoutAsEndOfMessage constructor argument to true.

Each of these is a subclass of AbstractByteArraySerializer, which implements both org.springframework.core.serializer.Serializer and org.springframework.core.serializer.Deserializer. For backwards compatibility, connections that use any subclass of AbstractByteArraySerializer for serialization also accept a String that is first converted to a byte array. Each of these serializers and deserializers converts an input stream that contains the corresponding format to a byte array payload.

To avoid memory exhaustion due to a badly behaved client (one that does not adhere to the protocol of the configured serializer), these serializers impose a maximum message size. If an incoming message exceeds this size, an exception is thrown. The default maximum message size is

2048 bytes. You can increase it by setting the `maxMessageSize` property. If you use the default serializer or deserializer and wish to increase the maximum message size, you must declare the maximum message size as an explicit bean with the `maxMessageSize` property set and configure the connection factory to use that bean.

The classes marked with * earlier in this section use an intermediate buffer and copy the decoded data to a final buffer of the correct size. Starting with version 4.3, you can configure these buffers by setting a `poolSize` property to let these raw buffers be reused instead of being allocated and discarded for each message, which is the default behavior. Setting the property to a negative value creates a pool that has no bounds. If the pool is bounded, you can also set the `poolWaitTimeout` property (in milliseconds), after which an exception is thrown if no buffer becomes available. It defaults to infinity. Such an exception causes the socket to be closed.

If you wish to use the same mechanism in custom deserializers, you can extend `AbstractPooledBufferByteArraySerializer` (instead of its super class, `AbstractByteArraySerializer`) and implement `doDeserialize()` instead of `deserialize()`. The buffer is automatically returned to the pool. `AbstractPooledBufferByteArraySerializer` also provides a convenient utility method: `copyToSizedArray()`.

Version 5.0 added the `ByteArrayElasticRawDeserializer`. This is similar to the deserializer side of `ByteArrayRawSerializer` above, except that it is not necessary to set a `maxMessageSize`. Internally, it uses a `ByteArrayOutputStream` that lets the buffer grow as needed. The client must close the socket in an orderly manner to signal end of message.

This deserializer should only be used when the peer is trusted; it is susceptible to a DoS attach due to out of memory conditions.

The `MapJsonSerializer` uses a Jackson `ObjectMapper` to convert between a `Map` and JSON. You can use this serializer in conjunction with a `MessageConvertingTcpMessageMapper` and a `MapMessageConverter` to transfer selected headers and the payload in JSON.

The Jackson `ObjectMapper` cannot demarcate messages in the stream. Therefore, the `MapJsonSerializer` needs to delegate to another serializer or deserializer to handle message demarcation. By default, a `ByteArrayLfSerializer` is used, resulting in messages with a format of `<json><LF>` on the wire, but you can configure it to use others instead. (The next example shows how to do so.)

The final standard serializer is `org.springframework.core.serializer.DefaultSerializer`, which you can use to convert serializable objects with Java serialization. `org.springframework.core.serializer.DefaultDeserializer` is provided for inbound deserialization of streams that contain serializable objects.

If you do not wish to use the default serializer and deserializer (`ByteArrayCrLfSerializer`), you must set the `serializer` and `deserializer` attributes on the connection factory. The following example shows how to do so:

```
<bean id="javaSerializer"
      class="org.springframework.core.serializer.DefaultSerializer" />
<bean id="javaDeserializer"
      class="org.springframework.core.serializer.DefaultDeserializer" />

<int-ip:tcp-connection-factory id="server"
    type="server"
    port="1234"
    deserializer="javaDeserializer"
    serializer="javaSerializer"/>
```

A server connection factory that uses `java.net.Socket` connections and uses Java serialization on the wire.

For full details of the attributes available on connection factories, see the reference at the end of this section.

By default, reverse DNS lookups are done on inbound packets to convert IP addresses to host names for use in message headers. In environments where DNS is not configured, this can cause connection delays. You can override this default behavior by setting the `lookup-host` attribute to `false`.

You can also modify the attributes of sockets and socket factories. See SSL/TLS Support for more information. As noted there, such modifications are possible whether or not SSL is being used.

36.3.3. Custom Serializers and Deserializers

If your data is not in a format supported by one of the standard deserializers, you can implement your own; you can also implement a custom serializer.

To implement a custom serializer and deserializer pair, implement the `org.springframework.core.serializer.Deserializer` and `org.springframework.core.serializer.Serializer` interfaces.

When the deserializer detects a closed input stream between messages, it must throw a `SoftEndOfStreamException`; this is a signal to the framework to indicate that the close was "normal". If the stream is closed while decoding a message, some other exception should be thrown instead.

Starting with version 5.2, `SoftEndOfStreamException` is now a `RuntimeException` instead of extending `IOException`.

36.3.4. TCP Caching Client Connection Factory

As noted earlier, TCP sockets can be 'single-use' (one request or response) or shared. Shared sockets do not perform well with outbound gateways in high-volume environments, because the socket can only process one request or response at a time.

To improve performance, you can use collaborating channel adapters instead of gateways, but that requires application-level message correlation. See TCP Message Correlation for more information.

Spring Integration 2.2 introduced a caching client connection factory, which uses a pool of shared sockets, letting a gateway process multiple concurrent requests with a pool of shared connections.

36.3.5. TCP Failover Client Connection Factory

You can configure a TCP connection factory that supports failover to one or more other servers. When sending a message, the factory iterates over all its configured factories until either the message can be sent or no connection can be found. Initially, the first factory in the configured list is used. If a connection subsequently fails, the next factory becomes the current factory. The following example shows how to configure a failover client connection factory:

```
<bean id="failCF" class="o.s.i.ip.tcp.connection.FailoverClientConnectionFactory">
    <constructor-arg>
        <list>
            <ref bean="clientFactory1"/>
            <ref bean="clientFactory2"/>
        </list>
    </constructor-arg>
</bean>
```

When using the failover connection factory, the `singleUse` property must be consistent between the factory itself and the list of factories it is configured to use.

The connection factory has two properties related to failing back, when used with a shared connection (`singleUse=false`):

- `refreshSharedInterval`
- `closeOnRefresh`

Consider the following scenario based on the above configuration: Let's say `clientFactory1` cannot establish a connection but `clientFactory2` can. When the `failCF getConnection()` method is called after the `refreshSharedInterval` has passed, we will again attempt to connect using `clientFactory1`; if successful, the connection to `clientFactory2` will be closed. If `closeOnRefresh` is `false`, the "old" connection will remain open and may be reused in future if the first factory fails once more.

Set `refreshSharedInterval` to only attempt to reconnect with the first factory after that time has expired; setting it to `Long.MAX_VALUE` (default) if you only want to fail back to the first factory when the current connection fails.

Set `closeOnRefresh` to close the "old" connection after a refresh actually creates a new connection.

These properties do not apply if any of the delegate factories is a `CachingClientConnectionFactory` because the connection caching is handled there; in that case the list of connection factories will always be consulted to get a connection.

Starting with version 5.3, these default to `Long.MAX_VALUE` and `true` so the factory only attempts to fail back when the current connection fails. To revert to the default behavior of previous versions, set them to `0` and `false`.

Also see Testing Connections.

36.3.6. TCP Thread Affinity Connection Factory

Spring Integration version 5.0 introduced this connection factory. It binds a connection to the calling thread, and the same connection is reused each time that thread sends a message. This continues until the connection is closed (by the server or the network) or until the thread calls the `releaseConnection()` method. The connections themselves are provided by another client factory implementation, which must be configured to provide non-shared (single-use) connections so that each thread gets a connection.

The following example shows how to configure a TCP thread affinity connection factory:

```
@Bean
public TcpNetClientConnectionFactory cf() {
    TcpNetClientConnectionFactory cf = new TcpNetClientConnectionFactory(
"localhost",
            Integer.parseInt(System.getProperty(PORT)));
    cf.setSingleUse(true);
    return cf;
}

@Bean
public ThreadAffinityClientConnectionFactory tacf() {
    return new ThreadAffinityClientConnectionFactory(cf());
}

@Bean
@ServiceActivator(inputChannel = "out")
public TcpOutboundGateway outGate() {
    TcpOutboundGateway outGate = new TcpOutboundGateway();
    outGate.setConnectionFactory(tacf());
    outGate.setReplyChannelName("toString");
    return outGate;
}
```

36.4. Testing Connections

In some scenarios, it can be useful to send some kind of health-check request when a connection is first opened. One such scenario might be when using a TCP Failover Client Connection Factory so that we can fail over if the selected server allowed a connection to be opened but reports that it is not healthy.

In order to support this feature, add a `connectionTest` to the client connection factory.

```
/**
 * Set a {@link Predicate} that will be invoked to test a new connection; return
true
 * to accept the connection, false the reject.
 * @param connectionTest the predicate.
 * @since 5.3
 */
public void setConnectionTest(@Nullable Predicate<TcpConnectionSupport>
connectionTest) {
    this.connectionTest = connectionTest;
}
```

To test the connection, attach a temporary listener to the connection within the test. If the test fails, the connection is closed and an exception thrown. When used with the TCP Failover Client Connection Factory this triggers trying the next server.

Only the first reply from the server will go to the test listener.

In the following example, the server is considered healthy if the server replies `PONG` when we send `PING`.

```
Message<String> ping = new GenericMessage<>("PING");
byte[] pong = "PONG".getBytes();
clientFactory.setConnectionTest(conn -> {
    CountDownLatch latch = new CountDownLatch(1);
    AtomicBoolean result = new AtomicBoolean();
    conn.registerTestListener(msg -> {
        if (Arrays.equals(pong, (byte[]) msg.getPayload())) {
            result.set(true);
        }
        latch.countDown();
        return false;
    });
    conn.send(ping);
    try {
        latch.await(10, TimeUnit.SECONDS);
    }
    catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    return result.get();
});
```

36.5. TCP Connection Interceptors

You can configure connection factories with a reference to a `TcpConnectionInterceptorFactoryChain`. You can use interceptors to add behavior to connections, such as negotiation, security, and other options. No interceptors are currently provided by the framework, but see `InterceptedSharedConnectionTests` in the source repository for an example.

The `HelloWorldInterceptor` used in the test case works as follows:

The interceptor is first configured with a client connection factory. When the first message is sent over an intercepted connection, the interceptor sends 'Hello' over the connection and expects to receive 'world!'. When that occurs, the negotiation is complete and the original message is sent. Further messages that use the same connection are sent without any additional negotiation.

When configured with a server connection factory, the interceptor requires the first message to be 'Hello' and, if it is, returns 'world!'. Otherwise it throws an exception that causes the connection to be closed.

All `TcpConnection` methods are intercepted. Interceptor instances are created for each connection by an interceptor factory. If an interceptor is stateful, the factory should create a new instance for each connection. If there is no state, the same interceptor can wrap each connection. Interceptor factories are added to the configuration of an interceptor factory chain, which you can provide to a connection factory by setting the `interceptor-factory` attribute. Interceptors must extend `TcpConnectionInterceptorSupport`. Factories must implement the `TcpConnectionInterceptorFactory` interface. `TcpConnectionInterceptorSupport` has passthrough methods. By extending this class, you

only need to implement those methods you wish to intercept.

The following example shows how to configure a connection interceptor factory chain:

```
<bean id="helloWorldInterceptorFactory"
    class="o.s.i.ip.tcp.connection.TcpConnectionInterceptorFactoryChain">
    <property name="interceptors">
        <array>
            <bean class="o.s.i.ip.tcp.connection.HelloWorldInterceptorFactory"/>
        </array>
    </property>
</bean>

<int-ip:tcp-connection-factory id="server"
    type="server"
    port="12345"
    using-nio="true"
    single-use="true"
    interceptor-factory-chain="helloWorldInterceptorFactory"/>

<int-ip:tcp-connection-factory id="client"
    type="client"
    host="localhost"
    port="12345"
    single-use="true"
    so-timeout="100000"
    using-nio="true"
    interceptor-factory-chain="helloWorldInterceptorFactory"/>
```

36.6. TCP Connection Events

Beginning with version 3.0, changes to `TcpConnection` instances are reported by `TcpConnectionEvent` instances. `TcpConnectionEvent` is a subclass of `ApplicationEvent` and can thus be received by any `ApplicationListener` defined in the `ApplicationContext` — for example an event inbound channel adapter.

`TcpConnectionEvents` have the following properties:

- `connectionId`: The connection identifier, which you can use in a message header to send data to the connection.
- `connectionFactoryName`: The bean name of the connection factory to which the connection belongs.
- `throwable`: The `Throwable` (for `TcpConnectionExceptionEvent` events only).
- `source`: The `TcpConnection`. You can use this, for example, to determine the remote IP Address with `getHostAddress()` (cast required).

In addition, since version 4.0, the standard deserializers discussed in TCP Connection Factories now emit `TcpDeserializationExceptionEvent` instances when they encounter problems while decoding the data stream. These events contain the exception, the buffer that was in the process of being built, and an offset into the buffer (if available) at the point where the exception occurred. Applications can use a normal `ApplicationListener` or an `ApplicationEventListeningMessageProducer` (see Receiving Spring Application Events) to capture these events, allowing analysis of the problem.

Starting with versions 4.0.7 and 4.1.3, `TcpConnectionServerExceptionEvent` instances are published whenever an unexpected exception occurs on a server socket (such as a `BindException` when the server socket is in use). These events have a reference to the connection factory and the cause.

Starting with version 4.2, `TcpConnectionFailedCorrelationEvent` instances are published whenever an endpoint (inbound gateway or collaborating outbound channel adapter) receives a message that cannot be routed to a connection because the `ip_connectionId` header is invalid. Outbound gateways also publish this event when a late reply is received (the sender thread has timed out). The event contains the connection ID as well as an exception in the `cause` property, which contains the failed message.

Starting with version 4.3, a `TcpConnectionServerListeningEvent` is emitted when a server connection factory is started. This is useful when the factory is configured to listen on port 0, meaning that the operating system chooses the port. It can also be used instead of polling `isListening()`, if you need to wait before starting some other process that connects to the socket.

To avoid delaying the listening thread from accepting connections, the event is published on a separate thread.

Starting with version 4.3.2, a `TcpConnectionFailedEvent` is emitted whenever a client connection cannot be created. The source of the event is the connection factory, which you can use to determine the host and port to which the connection could not be established.

36.7. TCP Adapters

TCP inbound and outbound channel adapters that use connection factories mentioned earlier are provided. These adapters have two relevant attributes: `connection-factory` and `channel`. The `connection-factory` attribute indicates which connection factory is to be used to manage connections for the adapter. The `channel` attribute specifies the channel on which messages arrive at an outbound adapter and on which messages are placed by an inbound adapter. While both inbound and outbound adapters can share a connection factory, server connection factories are always “owned” by an inbound adapter. Client connection factories are always “owned” by an outbound adapter. Only one adapter of each type may get a reference to a connection factory. The following example shows how to define client and server TCP connection factories:

```
<bean id="javaSerializer"
      class="org.springframework.core.serializer.DefaultSerializer"/>
<bean id="javaDeserializer"
      class="org.springframework.core.serializer.DefaultDeserializer"/>

<int-ip:tcp-connection-factory id="server"
    type="server"
    port="1234"
    deserializer="javaDeserializer"
    serializer="javaSerializer"
    using-nio="true"
    single-use="true"/>

<int-ip:tcp-connection-factory id="client"
    type="client"
    host="localhost"
    port="#{server.port}"
    single-use="true"
    so-timeout="10000"
    deserializer="javaDeserializer"
    serializer="javaSerializer"/>

<int:channel id="input" />

<int:channel id="replies">
    <int:queue/>
</int:channel>

<int-ip:tcp-outbound-channel-adapter id="outboundClient"
    channel="input"
    connection-factory="client"/>

<int-ip:tcp-inbound-channel-adapter id="inboundClient"
    channel="replies"
    connection-factory="client"/>

<int-ip:tcp-inbound-channel-adapter id="inboundServer"
    channel="loop"
    connection-factory="server"/>

<int-ip:tcp-outbound-channel-adapter id="outboundServer"
    channel="loop"
    connection-factory="server"/>

<int:channel id="loop"/>
```

In the preceding configuration, messages arriving in the `input` channel are serialized over connections created by `client` connection factory, received at the server, and placed on the `loop`

channel. Since `loop` is the input channel for `outboundServer`, the message is looped back over the same connection, received by `inboundClient`, and deposited in the `replies` channel. Java serialization is used on the wire.

Normally, inbound adapters use a `type="server"` connection factory, which listens for incoming connection requests. In some cases, you may want to establish the connection in reverse, such that the inbound adapter connects to an external server and then waits for inbound messages on that connection.

This topology is supported by setting `client-mode="true"` on the inbound adapter. In this case, the connection factory must be of type `client` and must have `single-use` set to `false`.

Two additional attributes support this mechanism. The `retry-interval` specifies (in milliseconds) how often the framework attempts to reconnect after a connection failure. The `scheduler` supplies a `TaskScheduler` to schedule the connection attempts and to test that the connection is still active.

If you don't provide a scheduler, the framework's default taskScheduler bean is used.

For an outbound adapter, the connection is normally established when the first message is sent. The `client-mode="true"` on an outbound adapter causes the connection to be established when the adapter is started. By default, adapters are automatically started. Again, the connection factory must be of type `client` and have `single-use="false"`. A `retry-interval` and `scheduler` are also supported. If a connection fails, it is re-established either by the scheduler or when the next message is sent.

For both inbound and outbound, if the adapter is started, you can force the adapter to establish a connection by sending a `<control-bus />` command: `@adapter_id.retryConnection()`. Then you can examine the current state with `@adapter_id.isClientModeConnected()`.

36.8. TCP Gateways

The inbound TCP gateway `TcpInboundGateway` and outbound TCP gateway `TcpOutboundGateway` use a server and client connection factory, respectively. Each connection can process a single request or response at a time.

The inbound gateway, after constructing a message with the incoming payload and sending it to the `requestChannel`, waits for a response and sends the payload from the response message by writing it to the connection.

For the inbound gateway, you must retain or populate, the `ip_connectionId` header, because it is used to correlate the message to a connection. Messages that originate at the gateway automatically have the header set. If the reply is constructed as a new message, you need to set the header. The header value can be captured from the incoming message.

As with inbound adapters, inbound gateways normally use a `type="server"` connection factory, which listens for incoming connection requests. In some cases, you may want to establish the connection in reverse, such that the inbound gateway connects to an external server and then waits for and replies to inbound messages on that connection.

This topology is supported by using `client-mode="true"` on the inbound gateway. In this case, the connection factory must be of type `client` and must have `single-use` set to `false`.

Two additional attributes support this mechanism. `retry-interval` specifies (in milliseconds) how often the framework tries to reconnect after a connection failure. `scheduler` supplies a `TaskScheduler` to schedule the connection attempts and to test that the connection is still active.

If the gateway is started, you may force the gateway to establish a connection by sending a `<control-bus/>` command: `@adapter_id.retryConnection()` and examine the current state with `@adapter_id.isClientModeConnected()`.

The outbound gateway, after sending a message over the connection, waits for a response, constructs a response message, and puts it on the reply channel. Communications over the connections are single-threaded. Only one message can be handled at a time. If another thread attempts to send a message before the current response has been received, it blocks until any previous requests are complete (or time out). If, however, the client connection factory is configured for single-use connections, each new request gets its own connection and is processed immediately. The following example configures an inbound TCP gateway:

```
<int-ip:tcp-inbound-gateway id="inGateway"
    request-channel="tcpChannel"
    reply-channel="replyChannel"
    connection-factory="cfServer"
    reply-timeout="10000"/>
```

If a connection factory configured with the default serializer or deserializer is used, messages is `\r\n` delimited data and the gateway can be used by a simple client such as telnet.

The following example shows an outbound TCP gateway:

```
<int-ip:tcp-outbound-gateway id="outGateway"
    request-channel="tcpChannel"
    reply-channel="replyChannel"
    connection-factory="cfClient"
    request-timeout="10000"
    remote-timeout="10000"/> <!-- or e.g.
remote-timeout-expression="headers['timeout']" -->
```

`client-mode` is not currently available with the outbound gateway.

Starting with version 5.2, the outbound gateway can be configured with the property `closeStreamAfterSend`. If the connection factory is configured for `single-use` (a new connection for each request/reply) the gateway will close the output stream; this signals EOF to the server. This is useful if the server uses the EOF to determine the end of message, rather than some delimiter in the stream, but leaves the connection open in order to receive the reply.

Normally, the calling thread will block in the gateway, waiting for the reply (or a timeout). Starting with version 5.3, you can set the `async` property on the gateway and the sending thread is released to do other work. The reply (or error) will be sent on the receiving thread. This only applies when using the `TcpNetClientConnectionFactory`, it is ignored when using NIO because there is a race condition whereby a socket error that occurs after the reply is received can be passed to the gateway before the reply.

When using a shared connection (`singleUse=false`), a new request, while another is in process, will be blocked until the current reply is received. Consider using the `CachingClientConnectionFactory` if you wish to support concurrent requests on a pool of long-lived connections.

36.9. TCP Message Correlation

One goal of the IP endpoints is to provide communication with systems other than Spring Integration applications. For this reason, only message payloads are sent and received by default. Since 3.0, you can transfer headers by using JSON, Java serialization, or custom serializers and deserializers. See Transferring Headers for more information. No message correlation is provided by the framework (except when using the gateways) or collaborating channel adapters on the server side. Later in this document, we discuss the various correlation techniques available to applications. In most cases, this requires specific application-level correlation of messages, even when message payloads contain some natural correlation data (such as an order number).

36.9.1. Gateways

Gateways automatically correlate messages. However, you should use an outbound gateway for relatively low-volume applications. When you configure the connection factory to use a single shared connection for all message pairs ('single-use="false"'), only one message can be processed at a time. A new message has to wait until the reply to the previous message has been received. When a connection factory is configured for each new message to use a new connection ('single-use="true"'), this restriction does not apply. While this setting can give higher throughput than a shared connection environment, it comes with the overhead of opening and closing a new connection for each message pair.

Therefore, for high-volume messages, consider using a collaborating pair of channel adapters. However, to do so, you need to provide collaboration logic.

Another solution, introduced in Spring Integration 2.2, is to use a `CachingClientConnectionFactory`, which allows the use of a pool of shared connections.

36.9.2. Collaborating Outbound and Inbound Channel Adapters

To achieve high-volume throughput (avoiding the pitfalls of using gateways, as mentioned earlier) you can configure a pair of collaborating outbound and inbound channel adapters. You can also use collaborating adapters (server-side or client-side) for totally asynchronous communication (rather than with request-reply semantics). On the server side, message correlation is automatically handled by the adapters, because the inbound adapter adds a header that allows the outbound adapter to determine which connection to use when sending the reply message.

On the server side, you must populate the `ip_connectionId` header, because it is used to correlate the message to a connection. Messages that originate at the inbound adapter automatically have the header set. If you wish to construct other messages to send, you need to set the header. You can get the header value from an incoming message.

On the client side, the application must provide its own correlation logic, if needed. You can do so in a number of ways.

If the message payload has some natural correlation data (such as a transaction ID or an order number) and you have no need to retain any information (such as a reply channel header) from the original outbound message, the correlation is simple and would be done at the application level in any case.

If the message payload has some natural correlation data (such as a transaction ID or an order number), but you need to retain some information (such as a reply channel header) from the original outbound message, you can retain a copy of the original outbound message (perhaps by using a publish-subscribe channel) and use an aggregator to recombine the necessary data.

For either of the previous two scenarios, if the payload has no natural correlation data, you can provide a transformer upstream of the outbound channel adapter to enhance the payload with such data. Such a transformer may transform the original payload to a new object that contains both the original payload and some subset of the message headers. Of course, live objects (such as reply channels) from the headers cannot be included in the transformed payload.

If you choose such a strategy, you need to ensure the connection factory has an appropriate serializer-deserializer pair to handle such a payload (such as `DefaultSerializer` and `DefaultDeserializer`, which use java serialization, or a custom serializer and deserializer). The `ByteArray*Serializer` options mentioned in TCP Connection Factories, including the default `ByteArrayCrLfSerializer`, do not support such payloads unless the transformed payload is a `String` or `byte[]`.

Before the 2.2 release, when collaborating channel adapters used a client connection factory, the `so-timeout` attribute defaulted to the default reply timeout (10 seconds). This meant that, if no data were received by the inbound adapter for this period of time, the socket was closed.

This default behavior was not appropriate in a truly asynchronous environment, so it now defaults to an infinite timeout. You can reinstate the previous default behavior by setting the `so-timeout` attribute on the client connection factory to 10000 milliseconds.

36.9.3. Transferring Headers

TCP is a streaming protocol. `Serializers` and `Deserializers` demarcate messages within the stream. Prior to 3.0, only message payloads (`String` or `byte[]`) could be transferred over TCP. Beginning with 3.0, you can transfer selected headers as well as the payload. However, “live” objects, such as the `replyChannel` header, cannot be serialized.

Sending header information over TCP requires some additional configuration.

The first step is to provide the `ConnectionFactory` with a `MessageConvertingTcpMessageMapper` that uses the `mapper` attribute. This mapper delegates to any `MessageConverter` implementation to convert the message to and from some object that can be serialized and deserialized by the configured `serializer` and `deserializer`.

Spring Integration provides a `MapMessageConverter`, which allows the specification of a list of headers that are added to a `Map` object, along with the payload. The generated Map has two entries: `payload` and `headers`. The `headers` entry is itself a `Map` and contains the selected headers.

The second step is to provide a serializer and a deserializer that can convert between a `Map` and some wire format. This can be a custom `Serializer` or `Deserializer`, which you typically need if the peer system is not a Spring Integration application.

Spring Integration provides a `MapJsonSerializer` to convert a `Map` to and from JSON. It uses a Spring Integration `JsonObjectMapper`. You can provide a custom `JsonObjectMapper` if needed. By default, the serializer inserts a linefeed (`0x0a`) character between objects. See the Javadoc for more information.

The `JsonObjectMapper` uses whichever version of `Jackson` is on the classpath.

You can also use standard Java serialization of the `Map`, by using the `DefaultSerializer` and `DefaultDeserializer`.

The following example shows the configuration of a connection factory that transfers the `correlationId`, `sequenceNumber`, and `sequenceSize` headers by using JSON:

```
<int-ip:tcp-connection-factory id="client"
    type="client"
    host="localhost"
    port="12345"
    mapper="mapper"
    serializer="jsonSerializer"
    deserializer="jsonSerializer"/>

<bean id="mapper"
      class="o.sf.integration.ip.tcp.connection.MessageConvertingTcpMessageMapper
">
    <constructor-arg name="messageConverter">
        <bean class="o.sf.integration.support.converter.MapMessageConverter">
            <property name="headerNames">
                <list>
                    <value>correlationId</value>
                    <value>sequenceNumber</value>
                    <value>sequenceSize</value>
                </list>
            </property>
        </bean>
    </constructor-arg>
</bean>

<bean id="jsonSerializer" class=
"o.sf.integration.ip.tcp.serializer.MapJsonSerializer" />
```

A message sent with the preceding configuration, with a payload of 'something' would appear on the wire as follows:

```
{"headers":{"correlationId":"things","sequenceSize":5,"sequenceNumber":1},"payload
":"something"}
```

36.10. About Non-blocking I/O (NIO)

Using NIO (see `using-nio` in IP Configuration Attributes) avoids dedicating a thread to read from each socket. For a small number of sockets, you are likely to find that not using NIO, together with an asynchronous hand-off (such as to a `QueueChannel`), performs as well as or better than using NIO.

You should consider using NIO when handling a large number of connections. However, the use of NIO has some other ramifications. A pool of threads (in the task executor) is shared across all the sockets. Each incoming message is assembled and sent to the configured channel as a separate unit of work on a thread selected from that pool. Two sequential messages arriving on the same socket might be processed by different threads. This means that the order in which the messages are sent

to the channel is indeterminate. Strict ordering of the messages arriving on the socket is not maintained.

For some applications, this is not an issue. For others, it is a problem. If you require strict ordering, consider setting `using-nio` to `false` and using an asynchronous hand-off.

Alternatively, you can insert a resequencer downstream of the inbound endpoint to return the messages to their proper sequence. If you set `apply-sequence` to `true` on the connection factory, messages arriving on a TCP connection have `sequenceNumber` and `correlationId` headers set. The resequencer uses these headers to return the messages to their proper sequence.

Starting with version 5.1.4, priority is given to accepting new connections over reading from existing connections. This should, generally, have little impact unless you have a very high rate of new incoming connections. If you wish to revert to the previous behavior of giving reads priority, set the `multiAccept` property on the `TcpNioServerConnectionFactory` to `false`.

36.10.1. Pool Size

The pool size attribute is no longer used. Previously, it specified the size of the default thread pool when a task-executor was not specified. It was also used to set the connection backlog on server sockets. The first function is no longer needed (see the next paragraph). The second function is replaced by the `backlog` attribute.

Previously, when using a fixed thread pool task executor (which was the default) with NIO, it was possible to get a deadlock and processing would stop. The problem occurred when a buffer was full, a thread reading from the socket was trying to add more data to the buffer, and no threads were available to make space in the buffer. This only occurred with a very small pool size, but it could be possible under extreme conditions. Since 2.2, two changes have eliminated this problem. First, the default task executor is a cached thread pool executor. Second, deadlock detection logic has been added such that, if thread starvation occurs, instead of deadlocking, an exception is thrown, thus releasing the deadlocked resources.

Now that the default task executor is unbounded, it is possible that an out-of-memory condition might occur with high rates of incoming messages, if message processing takes extended time. If your application exhibits this type of behavior, you should use a pooled task executor with an appropriate pool size, but see the next section.

36.10.2. Thread Pool Task Executor with `CALLER_RUNS` Policy

You should keep in mind some important considerations when you use a fixed thread pool with the `CallerRunsPolicy` (`CALLER_RUNS` when using the `<task/>` namespace) and the queue capacity is small.

The following does not apply if you do not use a fixed thread pool.

With NIO connections, there are three distinct task types. The I/O selector processing is performed on one dedicated thread (detecting events, accepting new connections, and dispatching the I/O read operations to other threads by using the task executor). When an I/O reader thread (to which the

read operation is dispatched) reads data, it hands off to another thread to assemble the incoming message. Large messages can take several reads to complete. These “assembler” threads can block while waiting for data. When a new read event occurs, the reader determines if this socket already has an assembler and, if not, runs a new one. When the assembly process is complete, the assembler thread is returned to the pool.

This can cause a deadlock when the pool is exhausted, the `CALLER_RUNS` rejection policy is in use, and the task queue is full. When the pool is empty and there is no room in the queue, the IO selector thread receives an `OP_READ` event and dispatches the read by using the executor. The queue is full, so the selector thread itself starts the read process. Now it detects that there is no assembler for this socket and, before it does the read, fires off an assembler. Again, the queue is full, and the selector thread becomes the assembler. The assembler is now blocked, waiting for the data to be read, which never happens. The connection factory is now deadlocked because the selector thread cannot handle new events.

To avoid this deadlock, we must avoid the selector (or reader) threads performing the assembly task. We want to use separate pools for the IO and assembly operations.

The framework provides a `CompositeExecutor`, which allows the configuration of two distinct executors: one for performing IO operations and one for message assembly. In this environment, an IO thread can never become an assembler thread, and the deadlock cannot occur.

In addition, the task executors should be configured to use an `AbortPolicy` (`ABORT` when using `<task>`). When an I/O task cannot be completed, it is deferred for a short time and continually retried until it can be completed and have an assembler allocated. By default, the delay is 100ms, but you can change it by setting the `readDelay` property on the connection factory (`read-delay` when configuring with the XML namespace).

The following three examples shows how to configure the composite executor:

```
@Bean
private CompositeExecutor compositeExecutor() {
    ThreadPoolTaskExecutor ioExec = new ThreadPoolTaskExecutor();
    ioExec.setCorePoolSize(4);
    ioExec.setMaxPoolSize(10);
    ioExec.setQueueCapacity(0);
    ioExec.setThreadNamePrefix("io-");
    ioExec.setRejectedExecutionHandler(new AbortPolicy());
    ioExec.initialize();
    ThreadPoolTaskExecutor assemblerExec = new ThreadPoolTaskExecutor();
    assemblerExec.setCorePoolSize(4);
    assemblerExec.setMaxPoolSize(10);
    assemblerExec.setQueueCapacity(0);
    assemblerExec.setThreadNamePrefix("assembler-");
    assemblerExec.setRejectedExecutionHandler(new AbortPolicy());
    assemblerExec.initialize();
    return new CompositeExecutor(ioExec, assemblerExec);
}
```

```
<bean id="myTaskExecutor" class=
"org.springframework.integration.util.CompositeExecutor">
    <constructor-arg ref="io"/>
    <constructor-arg ref="assembler"/>
</bean>

<task:executor id="io" pool-size="4-10" queue-capacity="0" rejection-policy="
ABORT" />
<task:executor id="assembler" pool-size="4-10" queue-capacity="0" rejection-
policy="ABORT" />
```

```
<bean id="myTaskExecutor" class=
"org.springframework.integration.util.CompositeExecutor">
    <constructor-arg>
        <bean class=
"org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
            <property name="threadNamePrefix" value="io-" />
            <property name="corePoolSize" value="4" />
            <property name="maxPoolSize" value="8" />
            <property name="queueCapacity" value="0" />
            <property name="rejectedExecutionHandler">
                <bean class="java.util.concurrent.ThreadPoolExecutor.AbortPolicy"
/>
            </property>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean class=
"org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
            <property name="threadNamePrefix" value="assembler-" />
            <property name="corePoolSize" value="4" />
            <property name="maxPoolSize" value="10" />
            <property name="queueCapacity" value="0" />
            <property name="rejectedExecutionHandler">
                <bean class="java.util.concurrent.ThreadPoolExecutor.AbortPolicy"
/>
            </property>
        </bean>
    </constructor-arg>
</bean>
```

36.11. SSL/TLS Support

Secure Sockets Layer/Transport Layer Security is supported. When using NIO, the JDK 5+ `SSLEngine` feature is used to handle handshaking after the connection is established. When not using NIO, standard `SSLSocketFactory` and `SSLServerSocketFactory` objects are used to create connections. A number of strategy interfaces are provided to allow significant customization. The default implementations of these interfaces provide for the simplest way to get started with secure communications.

36.11.1. Getting Started

Regardless of whether you use NIO, you need to configure the `ssl-context-support` attribute on the connection factory. This attribute references a <bean/> definition that describes the location and passwords for the required key stores.

SSL/TLS peers require two key stores each:

- A keystore that contains private and public key pairs to identify the peer
- A truststore that contains the public keys for peers that are trusted. See the documentation for the `keytool` utility provided with the JDK. The essential steps are
  1. Create a new key pair and store it in a keystore.
  2. Export the public key.
  3. Import the public key into the peer's truststore.
  4. Repeat for the other peer.

It is common in test cases to use the same key stores on both peers, but this should be avoided for production.

After establishing the key stores, the next step is to indicate their locations to the `TcpSSLContextSupport` bean and provide a reference to that bean to the connection factory.

The following example configures an SSL connection:

```
<bean id="sslContextSupport"
    class="o.sf.integration.ip.tcp.connection.support.DefaultTcpSSLContextSupport
">
    <constructor-arg value="client.ks"/>
    <constructor-arg value="client.truststore.ks"/>
    <constructor-arg value="secret"/>
    <constructor-arg value="secret"/>
</bean>

<ip:tcp-connection-factory id="clientFactory"
    type="client"
    host="localhost"
    port="1234"
    ssl-context-support="sslContextSupport" />
```

The `DefaultTcpSSLContextSupport` class also has an optional `protocol` property, which can be `SSL` or `TLS` (the default).

The keystore file names (the first two constructor arguments) use the Spring `Resource` abstraction. By default, the files are located on the classpath, but you can override this by using the `file:` prefix (to find the files on the filesystem instead).

Starting with version 4.3.6, when you use NIO, you can specify an `ssl-handshake-timeout` (in seconds) on the connection factory. This timeout (the default is 30 seconds) is used during SSL handshake when waiting for data. If the timeout is exceeded, the process is aborted and the socket is closed.

36.11.2. Host Verification

Starting with version 5.0.8, you can configure whether or not to enable host verification. Starting with version 5.1, it is enabled by default; the mechanism to disable it depends on whether or not you are using NIO.

Host verification is used to ensure the server you are connected to matches information in the certificate, even if the certificate is trusted.

When using NIO, configure the `DefaultTcpNioSSLConnectionSupport`, for example.

```
@Bean
public DefaultTcpNioSSLConnectionSupport connectionSupport() {
    DefaultTcpSSLContextSupport sslContextSupport = new
DefaultTcpSSLContextSupport("test.ks",
            "test.truststore.ks", "secret", "secret");
    sslContextSupport.setProtocol("SSL");
    DefaultTcpNioSSLConnectionSupport tcpNioConnectionSupport =
            new DefaultTcpNioSSLConnectionSupport(sslContextSupport, false);
    return tcpNioConnectionSupport;
}
```

The second constructor argument disables host verification. The `connectionSupport` bean is then injected into the NIO connection factory.

When not using NIO, the configuration is in the `TcpSocketSupport`:

```
connectionFactory.setTcpSocketSupport(new DefaultTcpSocketSupport(false));
```

Again, the constructor argument disables host verification.

36.12. Advanced Techniques

This section covers advanced techniques that you may find to be helpful in certain situations.

36.12.1. Strategy Interfaces

In many cases, the configuration described earlier is all that is needed to enable secure communication over TCP/IP. However, Spring Integration provides a number of strategy interfaces to allow customization and modification of socket factories and sockets:

- `TcpSSLContextSupport`
- `TcpSocketFactorySupport`
- `TcpSocketSupport`

- `TcpNetConnectionSupport`
- `TcpNioConnectionSupport`

The `TcpSSLContextSupport` Strategy Interface

The following listing shows the `TcpSSLContextSupport` strategy interface:

```
public interface TcpSSLContextSupport {

    SSLContext getSSLContext() throws Exception;

}
```

Implementations of the `TcpSSLContextSupport` interface are responsible for creating an `SSLContext` object. The implementation provided by the framework is the `DefaultTcpSSLContextSupport`, described earlier. If you require different behavior, implement this interface and provide the connection factory with a reference to a bean of your class' implementation.

The `TcpSocketFactorySupport` Strategy Interface

The following listing shows the definition of the `TcpSocketFactorySupport` strategy interface:

```
public interface TcpSocketFactorySupport {

    ServerSocketFactory getServerSocketFactory();

    SocketFactory getSocketFactory();

}
```

Implementations of this interface are responsible for obtaining references to `ServerSocketFactory` and `SocketFactory`. Two implementations are provided. The first is `DefaultTcpNetSocketFactorySupport` for non-SSL sockets (when no `ssl-context-support` attribute is defined). This uses the JDK's default factories. The second implementation is `DefaultTcpNetSSLSocketFactorySupport`. By default, this is used when an `ssl-context-support` attribute is defined. It uses the `SSLContext` created by that bean to create the socket factories.

This interface applies only if `using-nio` is `false`. NIO does not use socket factories.

The `TcpSocketSupport` Strategy Interface

The following listing shows the definition of the `TcpSocketSupport` strategy interface:

```
public interface TcpSocketSupport {

    void postProcessServerSocket(ServerSocket serverSocket);

    void postProcessSocket(Socket socket);

}
```

Implementations of this interface can modify sockets after they are created and after all configured attributes have been applied but before the sockets are used. This applies whether you use NIO or not. For example, you could use an implementation of this interface to modify the supported cipher suites on an SSL socket, or you could add a listener that gets notified after SSL handshaking is complete. The sole implementation provided by the framework is the `DefaultTcpSocketSupport`, which does not modify the sockets in any way.

To supply your own implementation of `TcpSocketFactorySupport` or `TcpSocketSupport`, provide the connection factory with references to beans of your custom type by setting the `socket-factory-support` and `socket-support` attributes, respectively.

The `TcpNetConnectionSupport` Strategy Interface

The following listing shows the definition of the `TcpNetConnectionSupport` strategy interface:

```
public interface TcpNetConnectionSupport {

    TcpNetConnection createNewConnection(Socket socket,
            boolean server, boolean lookupHost,
            ApplicationEventPublisher applicationEventPublisher,
            String connectionFactoryName) throws Exception;

}
```

This interface is invoked to create objects of type `TcpNetConnection` (or its subclasses). The framework provides a single implementation (`DefaultTcpNetConnectionSupport`), which, by default, creates simple `TcpNetConnection` objects. It has two properties: `pushbackCapable` and `pushbackBufferSize`. When push back is enabled, the implementation returns a subclass that wraps the connection's `InputStream` in a `PushbackInputStream`. Aligned with the `PushbackInputStream` default, the buffer size defaults to 1. This lets deserializers "unread" (push back) bytes into the stream. The following trivial example shows how it might be used in a delegating deserializer that "peeks" at the first byte to determine which deserializer to invoke:

```
public class CompositeDeserializer implements Deserializer<byte[]> {

    private final ByteArrayStxEtxSerializer stxEtx = new
ByteArrayStxEtxSerializer();

    private final ByteArrayCrLfSerializer crlf = new ByteArrayCrLfSerializer();

    @Override
    public byte[] deserialize(InputStream inputStream) throws IOException {
        PushbackInputStream pbis = (PushbackInputStream) inputStream;
        int first = pbis.read();
        if (first < 0) {
            throw new SoftEndOfStreamException();
        }
        pbis.unread(first);
        if (first == ByteArrayStxEtxSerializer.STX) {
            this.receivedStxEtx = true;
            return this.stxEtx.deserialize(pbis);
        }
        else {
            this.receivedCrLf = true;
            return this.crlf.deserialize(pbis);
        }
    }

}
```

The `TcpNioConnectionSupport` Strategy Interface

The following listing shows the definition of the `TcpNioConnectionSupport` strategy interface:

```
public interface TcpNioConnectionSupport {

    TcpNioConnection createNewConnection(SocketChannel socketChannel,
            boolean server, boolean lookupHost,
            ApplicationEventPublisher applicationEventPublisher,
            String connectionFactoryName) throws Exception;

}
```

This interface is invoked to create `TcpNioConnection` objects (or objects from subclasses). Spring Integration provides two implementations: `DefaultTcpNioSSLConnectionSupport` and `DefaultTcpNioConnectionSupport`. Which one is used depends on whether SSL is in use. A common use case is to subclass `DefaultTcpNioSSLConnectionSupport` and override `postProcessSSLEngine`. See the SSL client authentication example. As with the `DefaultTcpNetConnectionSupport`, these

implementations also support push back.

36.12.2. Example: Enabling SSL Client Authentication

To enable client certificate authentication when you use SSL, the technique depends on whether you use NIO. When you do not NIO , provide a custom `TcpSocketSupport` implementation to post-process the server socket:

```
serverFactory.setTcpSocketSupport(new DefaultTcpSocketSupport() {

    @Override
    public void postProcessServerSocket(ServerSocket serverSocket) {
        ((SSLServerSocket) serverSocket).setNeedClientAuth(true);
    }

});
```

(When you use XML configuration, provide a reference to your bean by setting the `socket-support` attribute).

When you use NIO, provide a custom `TcpNioSslConnectionSupport` implementation to post-process the `SSLEngine`, as the following example shows:

```
@Bean
public DefaultTcpNioSSLConnectionSupport tcpNioConnectionSupport() {
    return new DefaultTcpNioSSLConnectionSupport(serverSslContextSupport) {

            @Override
            protected void postProcessSSLEngine(SSLEngine sslEngine) {
                sslEngine.setNeedClientAuth(true);
            }

    }
}

@Bean
public TcpNioServerConnectionFactory server() {
    ...
    serverFactory.setTcpNioConnectionSupport(tcpNioConnectionSupport());
    ...
}
```

(When you use XML configuration, since version 4.3.7, provide a reference to your bean by setting the `nio-connection-support` attribute).

36.13. IP Configuration Attributes

The following table describes attributes that you can set to configure IP connections:

Table 11. Connection Factory Attributes

Attribute Name	Client?	Server?	Allowed Values	Attribute Description
`type`	Y	Y	client, server	Determines whether the connection factory is a client or a server.
`host`	Y	N		The host name or IP address of the destination.
`port`	Y	Y		The port.
`serializer`	Y	Y		An implementation of `Serializer` used to serialize the payload. Defaults to `ByteArrayCrLfSerializer`
`deserializer`	Y	Y		An implementation of `Deserializer` used to deserialize the payload. Defaults to `ByteArrayCrLfSerializer`
`using-nio`	Y	Y	`true`, `false`	Whether or not connection uses NIO. Refer to the `java.nio` package for more information. See About Non-blocking I/O (NIO). Default: `false`.
`using-direct-buffers`	Y	N	`true`, `false`	When using NIO, whether or not the connection uses direct buffers. Refer to the `java.nio.ByteBuffer` documentation for more information. Must be `false` if `using-nio` is `false`.
`apply-sequence`	Y	Y	`true`, `false`	When you use NIO, it may be necessary to resequence messages. When this attribute is set to `true`, `correlationId` and `sequenceNumber` headers are added to received messages. See About Non-blocking I/O (NIO). Default: `false`.
`so-timeout`	Y	Y		Defaults to `0` (infinity), except for server connection factories with `single-use="true"`. In that case, it defaults to the default reply timeout (10 seconds).
`so-send-buffer-size`	Y	Y		See `java.net.Socket. setSendBufferSize()`.
`so-receive-buffer-size`	Y	Y		See `java.net.Socket. setReceiveBufferSize()`.
`so-keep-alive`	Y	Y	`true`, `false`	See `java.net.Socket.setKeepAlive()`.

Attribute Name	Client?	Server?	Allowed Values	Attribute Description
so-linger	Y	Y		Sets linger to true with the supplied value. See java.net.Socket.setSoLinger().
so-tcp-no-delay	Y	Y	true, false	See java.net.Socket.setTcpNoDelay().
so-traffic-class	Y	Y		See java.net.Socket. setTrafficClass().
local-address	N	Y		On a multi-homed system, specifies an IP address for the interface to which the socket is bound.
task-executor	Y	Y		Specifies a specific executor to be used for socket handling. If not supplied, an internal cached thread executor is used. Needed on some platforms that require the use of specific task executors, such as a WorkManagerTaskExecutor.
single-use	Y	Y	true, false	Specifies whether a connection can be used for multiple messages. If true, a new connection is used for each message.
pool-size	N	N		This attribute is no longer used. For backward compatibility, it sets the backlog, but you should use backlog to specify the connection backlog in server factories.
backlog	N	Y		Sets the connection backlog for server factories.
lookup-host	Y	Y	true, false	Specifies whether reverse lookups are done on IP addresses to convert to host names for use in message headers. If false, the IP address is used instead. Default: true.
interceptor-factory-chain	Y	Y		See TCP Connection Interceptors.
ssl-context-support	Y	Y		See SSL/TLS Support.
socket-factory-support	Y	Y		See SSL/TLS Support.
socket-support	Y	Y		See SSL/TLS Support.
nio-connection-support	Y	Y		See Advanced Techniques.
read-delay	Y	Y	long > 0	The delay (in milliseconds) before retrying a read after the previous attempt failed due to insufficient threads. Default: 100. Only applies if using-nio is true.

The following table describes attributes that you can set to configure UDP inbound channel adapters:

Table 12. UDP Inbound Channel Adapter Attributes

Attribute Name	Allowed Values	Attribute Description
`port`		The port on which the adapter listens.
`multicast`	`true`, `false`	Whether or not the UDP adapter uses multicast.
`multicast-address`		When multicast is true, the multicast address to which the adapter joins.
`pool-size`		Specifies how many packets can be handled concurrently. It only applies if task-executor is not configured. Default: 5.
task-executor		Specifies a specific executor to be used for socket handling. If not supplied, an internal pooled executor is used. Needed on some platforms that require the use of specific task executors such as a `WorkManagerTaskExecutor`. See pool-size for thread requirements.
`receive-buffer-size`		The size of the buffer used to receive `DatagramPackets`. Usually set to the maximum transmission unit (MTU) size. If a smaller buffer is used than the size of the sent packet, truncation can occur. You can detect this by using the `check-length` attribute..
`check-length`	`true`, `false`	Whether or not a UDP adapter expects a data length field in the packet received. Used to detect packet truncation.
`so-timeout`		See the `setSoTimeout()` methods in `java.net.DatagramSocket` for more information.
`so-send-buffer-size`		Used for UDP acknowledgment packets. See the setSendBufferSize() methods in `java.net.DatagramSocket` for more information.
`so-receive-buffer-size`		See `java.net.DatagramSocket.setReceiveBufferSize()` for more information.
`local-address`		On a multi-homed system, specifies an IP address for the interface to which the socket is bound.
`error-channel`		If a downstream component throws an exception, the `MessagingException` message that contains the exception and failed message is sent to this channel.
`lookup-host`	`true`, `false`	Specifies whether reverse lookups are done on IP addresses to convert to host names for use in message headers. If `false`, the IP address is used instead. Default: `true`.

The following table describes attributes that you can set to configure UDP outbound channel adapters:

Table 13. UDP Outbound Channel Adapter Attributes

Attribute Name	Allowed Values	Attribute Description
`host`		The host name or ip address of the destination. For multicast udp adapters, the multicast address.
`port`		The port on the destination.
`multicast`	`true`, `false`	Whether or not the udp adapter uses multicast.
`acknowledge`	`true`, `false`	Whether or not a UDP adapter requires an acknowledgment from the destination. When enabled, it requires setting the following four attributes: `ack-host`, `ack-port`, `ack-timeout`, and `min-acks-for- success`.
`ack-host`		When `acknowledge` is `true`, indicates the host or IP address to which the acknowledgment should be sent. Usually the current host, but may be different — for example, when Network Address Translation (NAT) is being used.
`ack-port`		When `acknowledge` is `true`, indicates the port to which the acknowledgment should be sent. The adapter listens on this port for acknowledgments.
`ack-timeout`		When `acknowledge` is `true`, indicates the time in milliseconds that the adapter waits for an acknowledgment. If an acknowledgment is not received in time, the adapter throws an exception.
`min-acks-for- success`		Defaults to 1. For multicast adapters, you can set this to a larger value, which requires acknowledgments from multiple destinations.
`check-length`	`true`, `false`	Whether or not a UDP adapter includes a data length field in the packet sent to the destination.
`time-to-live`		For multicast adapters, specifies the time-to-live attribute for the `MulticastSocket`. Controls the scope of the multicasts. Refer to the Java API documentation for more information.
`so-timeout`		See `java.net.DatagramSocket` setSoTimeout() methods for more information.
`so-send-buffer-size`		See the `setSendBufferSize()` methods in `java.net.DatagramSocket` for more information.
`so-receive-buffer-size`		Used for UDP acknowledgment packets. See the `setReceiveBufferSize()` methods in `java.net.DatagramSocket` for more information.

Attribute Name	Allowed Values	Attribute Description
local-address		On a multi-homed system, for the UDP adapter, specifies an IP address for the interface to which the socket is bound for reply messages. For a multicast adapter, it also determines which interface the multicast packets are sent over.
`task-executor`		Specifies a specific executor to be used for acknowledgment handling. If not supplied, an internal single threaded executor is used. Needed on some platforms that require the use of specific task executors, such as a `WorkManagerTaskExecutor`. One thread is dedicated to handling acknowledgments (if the `acknowledge` option is true).
`destination-expression`	SpEL expression	A SpEL expression to be evaluated to determine which `SocketAddress` to use as a destination address for the outgoing UDP packets.
`socket-expression`	SpEL expression	A SpEL expression to be evaluated to determine which datagram socket use for sending outgoing UDP packets.

The following table describes attributes that you can set to configure TCP inbound channel adapters:

Table 14. TCP Inbound Channel Adapter Attributes

Attribute Name	Allowed Values	Attribute Description
`channel`		The channel to which inbound messages is sent.
`connection-factory`		If the connection factory has a type of `server`, the factory is “owned” by this adapter. If it has a type of `client`, it is “owned” by an outbound channel adapter, and this adapter receives any incoming messages on the connection created by the outbound adapter.
`error-channel`		If an exception is thrown by a downstream component, the `MessagingException` message containing the exception and the failed message is sent to this channel.
`client-mode`	`true`, `false`	When `true`, the inbound adapter acts as a client with respect to establishing the connection and then receiving incoming messages on that connection. Default: `false`. See also `retry-interval` and `scheduler`. The connection factory must be of type `client` and have `single-use` set to `false`.
`retry-interval`		When in `client-mode`, specifies the number of milliseconds to wait between connection attempts or after a connection failure. Default: 60000 (60 seconds).

Attribute Name	Allowed Values	Attribute Description
scheduler	true, false	Specifies a TaskScheduler to use for managing the client-mode connection. If not specified, it defaults to the global Spring Integration taskScheduler bean, which has a default pool size of 10. See Configuring the Task Scheduler.

The following table describes attributes that you can set to configure TCP outbound channel adapters:

Table 15. TCP Outbound Channel Adapter Attributes

Attribute Name	Allowed Values	Attribute Description
channel		The channel on which outbound messages arrive.
connection-factory		If the connection factory has a type of client, the factory is "owned" by this adapter. If it has a type of server, it is "owned" by an inbound channel adapter, and this adapter tries to correlate messages to the connection on which an original inbound message was received.
client-mode	true, false	When true, the outbound adapter tries to establish the connection as soon as it is started. When false, the connection is established when the first message is sent. Default: false. See also retry-interval and scheduler. The connection factory must be of type client and have single-use set to false.
retry-interval		When in client-mode, specifies the number of milliseconds to wait between connection attempts or after a connection failure. Default: 60000 (60 seconds).
scheduler	true, false	Specifies a TaskScheduler to use for managing the client-mode connection. If not specified, it defaults to the global Spring Integration taskScheduler bean, which has a default pool size of 10. See Configuring the Task Scheduler.

The following table describes attributes that you can set to configure TCP inbound gateways:

Table 16. TCP Inbound Gateway Attributes

Attribute Name	Allowed Values	Attribute Description
connection-factory		The connection factory must be of type server.
request-channel		The channel to which incoming messages are sent.

Attribute Name	Allowed Values	Attribute Description
`reply-channel`		The channel on which reply messages may arrive. Usually, replies arrive on a temporary reply channel added to the inbound message header.
`reply-timeout`		The time in milliseconds for which the gateway waits for a reply. Default: 1000 (1 second).
`error-channel`		If an exception is thrown by a downstream component, the `MessagingException` message containing the exception and the failed message is sent to this channel. Any reply from that flow is then returned as a response by the gateway.
`client-mode`	`true`, `false`	When `true`, the inbound gateway acts as a client with respect to establishing the connection and then receiving (and replying to) incoming messages on that connection. Default: false. See also `retry-interval` and `scheduler`. The connection factory must be of type `client` and have `single-use` set to `false`.
`retry-interval`		When in `client-mode`, specifies the number of milliseconds to wait between connection attempts or after a connection failure. Default: 60000 (60 seconds).
`scheduler`	`true`, `false`	Specifies a `TaskScheduler` to use for managing the `client-mode` connection. If not specified, it defaults to the global Spring Integration `taskScheduler` bean, which has a default pool size of 10. See Configuring the Task Scheduler.

The following table describes attributes that you can set to configure TCP outbound gateways:

Table 17. TCP Outbound Gateway Attributes

Attribute Name	Allowed Values	Attribute Description
`connection-factory`		The connection factory must be of type `client`.
`request-channel`		The channel on which outgoing messages arrive.
`reply-channel`		Optional. The channel to which reply messages are sent.
`remote-timeout`		The time in milliseconds for which the gateway waits for a reply from the remote system. Mutually exclusive with `remote-timeout-expression`. Default: 10000 (10 seconds). Note: In versions prior to 4.2 this value defaulted to `reply-timeout` (if set).

Attribute Name	Allowed Values	Attribute Description
remote-timeout-expression		A SpEL expression that is evaluated against the message to determine the time in milliseconds for which the gateway waits for a reply from the remote system. Mutually exclusive with remote-timeout.
request-timeout		If a single-use connection factory is not being used, the time in milliseconds for which the gateway waits to get access to the shared connection.
reply-timeout		The time in milliseconds for which the gateway waits when sending the reply to the reply-channel. Only applies if the reply-channel might block (such as a bounded QueueChannel that is currently full).
async		Release the sending thread after the send; the reply (or error) will be sent on the receiving thread.

36.14. IP Message Headers

IP Message Headers

This module uses the following MessageHeader instances:

Header Name	IpHeaders Constant	Description
ip_hostname	HOSTNAME	The host name from which a TCP message or UDP packet was received. If lookupHost is false, this contains the IP address.
ip_address	IP_ADDRESS	The IP address from which a TCP message or UDP packet was received.
ip_port	PORT	The remote port for a UDP packet.
ip_localInetAddress	IP_LOCAL_ADDRESS	The local InetAddress to which the socket is connected (since version 4.2.5).
ip_ackTo	ACKADDRESS	The remote IP address to which UDP application-level acknowledgments are sent. The framework includes acknowledgment information in the data packet.
ip_ackId	ACK_ID	A correlation ID for UDP application-level acknowledgments. The framework includes acknowledgment information in the data packet.
ip_tcp_remotePort	REMOTE_PORT	The remote port for a TCP connection.

Header Name	IpHeaders Constant	Description
ip_connectionId	CONNECTION_ID	A unique identifier for a TCP connection. Set by the framework for inbound messages. When sending to a server-side inbound channel adapter or replying to an inbound gateway, this header is required so that the endpoint can determine the connection to which to send the message.
ip_actualConnectionId	ACTUAL_CONNECTION_ID	For information only. When using a cached or failover client connection factory, it contains the actual underlying connection ID.
contentType	MessageHeaders. CONTENT_TYPE	An optional content type for inbound messages Described after this table. Note that, unlike the other header constants, this constant is in the MessageHeaders class, not the IpHeaders class.

For inbound messages, ip_hostname, ip_address, ip_tcp_remotePort, and ip_connectionId are mapped by the default TcpHeaderMapper. If you set the mapper's addContentTypeHeader property to true, the mapper sets the contentType header (application/octet-stream;charset="UTF-8", by default). You can change the default by setting the contentType property. You can add additional headers by subclassing TcpHeaderMapper and overriding the supplyCustomHeaders method. For example, when you use SSL, you can add properties of the SSLSession by obtaining the session object from the TcpConnection object, which is provided as an argument to the supplyCustomHeaders method.

For outbound messages, String payloads are converted to byte[] with the default (UTF-8) charset. Set the charset property to change the default.

When customizing the mapper properties or subclassing, declare the mapper as a bean and provide an instance to the connection factory by using the mapper property.

36.15. Annotation-Based Configuration

The following example from the samples repository shows some of the configuration options available when you use annotations instead of XML:

```
@EnableIntegration ①
@IntegrationComponentScan ②
@Configuration
public static class Config {

    @Value(${some.port})
    private int port;

    @MessagingGateway(defaultRequestChannel="toTcp") ③
    public interface Gateway {

        String viaTcp(String in);

    }

    @Bean
    @ServiceActivator(inputChannel="toTcp") ④
    public MessageHandler tcpOutGate(AbstractClientConnectionFactory
connectionFactory) {
        TcpOutboundGateway gate = new TcpOutboundGateway();
        gate.setConnectionFactory(connectionFactory);
        gate.setOutputChannelName("resultToString");
        return gate;
    }

    @Bean ⑤
    public TcpInboundGateway tcpInGate(AbstractServerConnectionFactory
connectionFactory)  {
        TcpInboundGateway inGate = new TcpInboundGateway();
        inGate.setConnectionFactory(connectionFactory);
        inGate.setRequestChannel(fromTcp());
        return inGate;
    }

    @Bean
    public MessageChannel fromTcp() {
        return new DirectChannel();
    }

    @MessageEndpoint
    public static class Echo { ⑥

        @Transformer(inputChannel="fromTcp", outputChannel="toEcho")
        public String convert(byte[] bytes) {
            return new String(bytes);
        }

        @ServiceActivator(inputChannel="toEcho")
        public String upCase(String in) {
```

```
            return in.toUpperCase();
        }

        @Transformer(inputChannel="resultToString")
        public String convertResult(byte[] bytes) {
            return new String(bytes);
        }

    }

    @Bean
    public AbstractClientConnectionFactory clientCF() { ⑦
        return new TcpNetClientConnectionFactory("localhost", this.port);
    }

    @Bean
    public AbstractServerConnectionFactory serverCF() { ⑧
        return new TcpNetServerConnectionFactory(this.port);
    }

}
```

① Standard Spring Integration annotation enabling the infrastructure for an integration application.

② Searches for `@MessagingGateway` interfaces.

③ The entry point to the client-side of the flow. The calling application can use `@Autowired` for this `Gateway` bean and invoke its method.

④ Outbound endpoints consist of a `MessageHandler` and a consumer that wraps it. In this scenario, the `@ServiceActivator` configures the endpoint, according to the channel type.

⑤ Inbound endpoints (in the TCP/UDP module) are all message-driven and so only need to be declared as simple `@Bean` instances.

⑥ This class provides a number of POJO methods for use in this sample flow (a `@Transformer` and `@ServiceActivator` on the server side and a `@Transformer` on the client side).

⑦ The client-side connection factory.

⑧ The server-side connection factory.

Chapter 37. WebFlux Support

The WebFlux Spring Integration module (`spring-integration-webflux`) allows for the execution of HTTP requests and the processing of inbound HTTP requests in a reactive manner.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-webflux</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-webflux:5.3.0.RC1"
```

The `io.projectreactor.netty:reactor-netty` dependency must be included in case of non-Servlet-based server configuration.

The WebFlux support consists of the following gateway implementations: `WebFluxInboundEndpoint` and `WebFluxRequestExecutingMessageHandler`. The support is fully based on the Spring WebFlux and Project Reactor foundations. See HTTP Support for more information, since many options are shared between reactive and regular HTTP components.

37.1. WebFlux Inbound Components

Starting with version 5.0, the `WebFluxInboundEndpoint` implementation of `WebHandler` is provided. This component is similar to the MVC-based `HttpRequestHandlingEndpointSupport`, with which it shares some common options through the newly extracted `BaseHttpInboundEndpoint`. It is used in the Spring WebFlux reactive environment (instead of MVC). The following example shows a simple implementation of a WebFlux endpoint:

```
@Configuration
@EnableWebFlux
@EnableIntegration
public class ReactiveHttpConfiguration {

    @Bean
    public WebFluxInboundEndpoint simpleInboundEndpoint() {
        WebFluxInboundEndpoint endpoint = new WebFluxInboundEndpoint();
        RequestMapping requestMapping = new RequestMapping();
        requestMapping.setPathPatterns("/test");
        endpoint.setRequestMapping(requestMapping);
        endpoint.setRequestChannelName("serviceChannel");
        return endpoint;
    }

    @ServiceActivator(inputChannel = "serviceChannel")
    String service() {
        return "It works!";
    }

}
```

The configuration is similar to the `HttpRequestHandlingEndpointSupport` (mentioned prior to the example), except that we use `@EnableWebFlux` to add the WebFlux infrastructure to our integration application. Also, the `WebFluxInboundEndpoint` performs `sendAndReceive` operations to the downstream flow by using back-pressure, on-demand based capabilities, provided by the reactive HTTP server implementation.

The reply part is non-blocking as well and is based on the internal `FutureReplyChannel`, which is flat-mapped to a reply `Mono` for on-demand resolution.

You can configure the `WebFluxInboundEndpoint` with a custom `ServerCodecConfigurer`, a `RequestedContentTypeResolver`, and even a `ReactiveAdapterRegistry`. The latter provides a mechanism you can use to return a reply as any reactive type: Reactor `Flux`, RxJava `Observable`, `Flowable`, and others. This way, we can implement Server Sent Events scenarios with Spring Integration components, as the following example shows:

```
@Bean
public IntegrationFlow sseFlow() {
    return IntegrationFlows
            .from(WebFlux.inboundGateway("/sse")
                    .requestMapping(m -> m.produces(MediaType
.TEXT_EVENT_STREAM_VALUE)))
            .handle((p, h) -> Flux.just("foo", "bar", "baz"))
            .get();
}
```

See Request Mapping Support and Cross-origin Resource Sharing (CORS) Support for more possible configuration options.

When the request body is empty or `payloadExpression` returns `null`, the request params (`MultiValueMap<String, String>`) is used for a `payload` of the target message to process.

37.1.1. Payload Validation

Starting with version 5.2, the `WebFluxInboundEndpoint` can be configured with a `Validator`. Unlike the MVC validation in the HTTP Support, it is used to validate elements in the `Publisher` to which a request has been converted by the `HttpMessageReader`, before performing a fallback and `payloadExpression` functions. The Framework can't assume how complex the `Publisher` object can be after building the final payload. If there is a requirements to restrict validation visibility for exactly final payload (or its `Publisher` elements), the validation should go downstream instead of WebFlux endpoint. See more information in the Spring WebFlux documentation. An invalid payload is rejected with an `IntegrationWebExchangeBindException` (a `WebExchangeBindException` extension), containing all the validation `Errors`. See more in Spring Framework Reference Manual about validation.

37.2. WebFlux Outbound Components

The `WebFluxRequestExecutingMessageHandler` (starting with version 5.0) implementation is similar to `HttpRequestExecutingMessageHandler`. It uses a `WebClient` from the Spring Framework WebFlux module. To configure it, define a bean similar to the following:

```
<bean id="httpReactiveOutbound"
    class=
"org.springframework.integration.webflux.outbound.WebFluxRequestExecutingMessageHa
ndler">
     <constructor-arg value="http://localhost:8080/example" />
     <property name="outputChannel" ref="responseChannel" />
</bean>
```

You can configure a `WebClient` instance to use, as the following example shows:

```
<beans:bean id="webClient" class=
"org.springframework.web.reactive.function.client.WebClient"
                factory-method="create"/>

<bean id="httpReactiveOutbound"
    class=
"org.springframework.integration.webflux.outbound.WebFluxRequestExecutingMessageHa
ndler">
     <constructor-arg value="http://localhost:8080/example" />
     <constructor-arg re="webClient" />
     <property name="outputChannel" ref="responseChannel" />
</bean>
```

The `WebClient exchange()` operation returns a `Mono<ClientResponse>`, which is mapped (by using several `Mono.map()` steps) to an `AbstractIntegrationMessageBuilder` as the output from the `WebFluxRequestExecutingMessageHandler`. Together with the `ReactiveChannel` as an `outputChannel`, the `Mono<ClientResponse>` evaluation is deferred until a downstream subscription is made. Otherwise, it is treated as an `async` mode, and the `Mono` response is adapted to a `SettableListenableFuture` for an asynchronous reply from the `WebFluxRequestExecutingMessageHandler`. The target payload of the output message depends on the `WebFluxRequestExecutingMessageHandler` configuration. The `setExpectedResponseType(Class<?>)` or `setExpectedResponseTypeExpression(Expression)` identifies the target type of the response body element conversion. If `replyPayloadToFlux` is set to `true`, the response body is converted to a `Flux` with the provided `expectedResponseType` for each element, and this `Flux` is sent as the payload downstream. Afterwards, you can use a splitter to iterate over this `Flux` in a reactive manner.

In addition a `BodyExtractor<?, ClientHttpResponse>` can be injected into the `WebFluxRequestExecutingMessageHandler` instead of the `expectedResponseType` and `replyPayloadToFlux` properties. It can be used for low-level access to the `ClientHttpResponse` and more control over body and HTTP headers conversion. Spring Integration provides `ClientHttpResponseBodyExtractor` as a identity function to produce (downstream) the whole `ClientHttpResponse` and any other possible custom logic.

Starting with version 5.2, the `WebFluxRequestExecutingMessageHandler` supports reactive `Publisher`, `Resource`, and `MultiValueMap` types as the request message payload. A respective `BodyInserter` is used internally to be populated into the `WebClient.RequestBodySpec`. When the payload is a reactive `Publisher`, a configured `publisherElementType` or `publisherElementTypeExpression` can be used to determine a type for the publisher's element type. The expression must be resolved to a `Class<?>`, `String` which is resolved to the target `Class<?>` or `ParameterizedTypeReference`.

See HTTP Outbound Components for more possible configuration options.

37.3. WebFlux Namespace Support

Spring Integration provides a `webflux` namespace and the corresponding schema definition. To include it in your configuration, add the following namespace declaration in your application

context configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-webflux="http://www.springframework.org/schema/integration/webflux"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    https://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/webflux
    https://www.springframework.org/schema/integration/webflux/spring-integration-
webflux.xsd">
    ...
</beans>
```

37.3.1. Inbound

To configure Spring Integration WebFlux with XML, you need to use appropriate components from the `int-webflux` namespace: `inbound-channel-adapter` or `inbound-gateway`, corresponding to request and response requirements, respectively. The following example shows how to configure both an inbound channel adapter and an inbound gateway:

```
<inbound-channel-adapter id="reactiveFullConfig" channel="requests"
                         path="test1"
                         auto-startup="false"
                         phase="101"
                         request-payload-type="byte[]"
                         error-channel="errorChannel"
                         payload-expression="payload"
                         supported-methods="PUT"
                         status-code-expression="'202'"
                         header-mapper="headerMapper"
                         codec-configurer="codecConfigurer"
                         reactive-adapter-registry="reactiveAdapterRegistry"
                         requested-content-type-resolver=
"requestedContentTypeResolver">
    <request-mapping headers="foo"/>
    <cross-origin origin="foo"
                  method="PUT"/>
    <header name="foo" expression="'foo'"/>
</inbound-channel-adapter>

<inbound-gateway id="reactiveFullConfig" request-channel="requests"
                 path="test1"
                 auto-startup="false"
                 phase="101"
                 request-payload-type="byte[]"
                 error-channel="errorChannel"
                 payload-expression="payload"
                 supported-methods="PUT"
                 reply-timeout-status-code-expression="'504'"
                 header-mapper="headerMapper"
                 codec-configurer="codecConfigurer"
                 reactive-adapter-registry="reactiveAdapterRegistry"
                 requested-content-type-resolver="requestedContentTypeResolver">
    <request-mapping headers="foo"/>
    <cross-origin origin="foo"
                  method="PUT"/>
    <header name="foo" expression="'foo'"/>
</inbound-gateway>
```

37.3.2. Outbound

If you want to execute the HTTP request in a reactive, non-blocking way, you can use the `outbound-gateway` or `outbound-channel-adapter`. The following example shows how to configure both an outbound gateway and an outbound channel adapter:

```
<int-webflux:outbound-gateway id="reactiveExample1"
    request-channel="requests"
    url="http://localhost/test"
    http-method-expression="headers.httpMethod"
    extract-request-payload="false"
    expected-response-type-expression="payload"
    charset="UTF-8"
    reply-timeout="1234"
    reply-channel="replies"/>

<int-webflux:outbound-channel-adapter id="reactiveExample2"
    url="http://localhost/example"
    http-method="GET"
    channel="requests"
    charset="UTF-8"
    extract-payload="false"
    expected-response-type="java.lang.String"
    order="3"
    auto-startup="false"/>
```

37.4. Configuring WebFlux Endpoints with Java

The following example shows how to configure a WebFlux inbound endpoint with Java:

```
@Bean
public WebFluxInboundEndpoint jsonInboundEndpoint() {
    WebFluxInboundEndpoint endpoint = new WebFluxInboundEndpoint();
    RequestMapping requestMapping = new RequestMapping();
    requestMapping.setPathPatterns("/persons");
    endpoint.setRequestMapping(requestMapping);
    endpoint.setRequestChannel(fluxResultChannel());
    return endpoint;
}

@Bean
public MessageChannel fluxResultChannel() {
    return new FluxMessageChannel();
}

@ServiceActivator(inputChannel = "fluxResultChannel")
Flux<Person> getPersons() {
    return Flux.just(new Person("Jane"), new Person("Jason"), new Person("John"));
}
```

The following example shows how to configure a WebFlux inbound gateway with the Java DSL:

```
@Bean
public IntegrationFlow inboundChannelAdapterFlow() {
    return IntegrationFlows
        .from(WebFlux.inboundChannelAdapter("/reactivePost")
            .requestMapping(m -> m.methods(HttpMethod.POST))
            .requestPayloadType(ResolvableType.forClassWithGenerics(Flux.class,
String.class))
            .statusCodeFunction(m -> HttpStatus.ACCEPTED))
        .channel(c -> c.queue("storeChannel"))
        .get();
}
```

The following example shows how to configure a WebFlux outbound gateway with Java:

```
@ServiceActivator(inputChannel = "reactiveHttpOutRequest")
@Bean
public WebFluxRequestExecutingMessageHandler reactiveOutbound(WebClient client) {
    WebFluxRequestExecutingMessageHandler handler =
        new WebFluxRequestExecutingMessageHandler("http://localhost:8080/foo",
client);
    handler.setHttpMethod(HttpMethod.POST);
    handler.setExpectedResponseType(String.class);
    return handler;
}
```

The following example shows how to configure a WebFlux outbound gateway with the Java DSL:

```
@Bean
public IntegrationFlow outboundReactive() {
    return f -> f
        .handle(WebFlux.<MultiValueMap<String, String>>outboundGateway(m ->
                UriComponentsBuilder.fromUriString("http://localhost:8080/foo")
                        .queryParams(m.getPayload())
                        .build()
                        .toUri())
                .httpMethod(HttpMethod.GET)
                .expectedResponseType(String.class));
}
```

37.5. WebFlux Header Mappings

Since WebFlux components are fully based on the HTTP protocol, there is no difference in the HTTP headers mapping. See HTTP Header Mappings for more possible options and components to use for mapping headers.

Chapter 38. WebSockets Support

Starting with version 4.1, Spring Integration has WebSocket support. It is based on the architecture, infrastructure, and API from the Spring Framework's `web-socket` module. Therefore, many of Spring WebSocket's components (such as `SubProtocolHandler` or `WebSocketClient`) and configuration options (such as `@EnableWebSocketMessageBroker`) can be reused within Spring Integration. For more information, see the Spring Framework WebSocket Support chapter in the Spring Framework reference manual.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-websocket</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-websocket:5.3.0.RC1"
```

For server side, the `org.springframework:spring-webmvc` dependency must be included explicitly.

The Spring Framework WebSocket infrastructure is based on the Spring messaging foundation and provides a basic messaging framework based on the same `MessageChannel` implementations and `MessageHandler` implementations that Spring Integration uses (and some POJO-method annotation mappings). Consequently, Spring Integration can be directly involved in a WebSocket flow, even without WebSocket adapters. For this purpose, you can configure a Spring Integration `@MessagingGateway` with appropriate annotations, as the following example shows:

```
@MessagingGateway
@Controller
public interface WebSocketGateway {

    @MessageMapping("/greeting")
    @SendToUser("/queue/answer")
    @Gateway(requestChannel = "greetingChannel")
    String greeting(String payload);

}
```

38.1. Overview

Since the WebSocket protocol is streaming by definition and we can send and receive messages to and from a WebSocket at the same time, we can deal with an appropriate `WebSocketSession`, regardless of being on the client or server side. To encapsulate the connection management and `WebSocketSession` registry, the `IntegrationWebSocketContainer` is provided with `ClientWebSocketContainer` and `ServerWebSocketContainer` implementations. Thanks to the WebSocket API and its implementation in the Spring Framework (with many extensions), the same classes are used on the server side as well as the client side (from a Java perspective, of course). Consequently, most connection and `WebSocketSession` registry options are the same on both sides. That lets us reuse many configuration items and infrastructure hooks to build WebSocket applications on the server side as well as on the client side. The following example shows how components can serve both purposes:

```
//Client side
@Bean
public WebSocketClient webSocketClient() {
    return new SockJsClient(Collections.singletonList(new WebSocketTransport(new
JettyWebSocketClient())));
}

@Bean
public IntegrationWebSocketContainer clientWebSocketContainer() {
    return new ClientWebSocketContainer(webSocketClient(),
"ws://my.server.com/endpoint");
}

//Server side
@Bean
public IntegrationWebSocketContainer serverWebSocketContainer() {
    return new ServerWebSocketContainer("/endpoint").withSockJs();
}
```

The `IntegrationWebSocketContainer` is designed to achieve bidirectional messaging and can be shared between inbound and outbound channel adapters (see below), can be referenced from only one of them when using one-way (sending or receiving) WebSocket messaging. It can be used without any channel adapter, but, in this case, `IntegrationWebSocketContainer` only plays a role as the `WebSocketSession` registry.

The `ServerWebSocketContainer` implements `WebSocketConfigurer` to register an internal `IntegrationWebSocketContainer.IntegrationWebSocketHandler` as an `Endpoint`. It does so under the provided `paths` and other server WebSocket options (such as `HandshakeHandler` or `SockJS fallback`) within the `ServletWebSocketHandlerRegistry` for the target vendor WebSocket Container. This registration is achieved with an infrastructural `WebSocketIntegrationConfigurationInitializer` component, which does the same as the `@EnableWebSocket` annotation. This means that, by using `@EnableIntegration` (or any Spring Integration namespace in the application context), you can omit the `@EnableWebSocket` declaration, because the Spring Integration infrastructure detects all WebSocket endpoints.

38.2. WebSocket Inbound Channel Adapter

The `WebSocketInboundChannelAdapter` implements the receiving part of `WebSocketSession` interaction. You must supply it with a `IntegrationWebSocketContainer`, and the adapter registers itself as a `WebSocketListener` to handle incoming messages and `WebSocketSession` events.

Only one `WebSocketListener` can be registered in the `IntegrationWebSocketContainer`.

For WebSocket subprotocols, the `WebSocketInboundChannelAdapter` can be configured with `SubProtocolHandlerRegistry` as the second constructor argument. The adapter delegates to the `SubProtocolHandlerRegistry` to determine the appropriate `SubProtocolHandler` for the accepted `WebSocketSession` and to convert a `WebSocketMessage` to a `Message` according to the sub-protocol implementation.

By default, the `WebSocketInboundChannelAdapter` relies only on the raw `PassThruSubProtocolHandler` implementation, which converts the `WebSocketMessage` to a `Message`.

The `WebSocketInboundChannelAdapter` accepts and sends to the underlying integration flow only `Message` instances that have `SimpMessageType.MESSAGE` or an empty `simpMessageType` header. All other `Message` types are handled through the `ApplicationEvent` instances emitted from a `SubProtocolHandler` implementation (such as `StompSubProtocolHandler`).

On the server side, if the `@EnableWebSocketMessageBroker` configuration is present, you can configure `WebSocketInboundChannelAdapter` with the `useBroker = true` option. In this case, all `non-MESSAGE Message` types are delegated to the provided `AbstractBrokerMessageHandler`. In addition, if the broker relay is configured with destination prefixes, those messages that match the Broker destinations are routed to the `AbstractBrokerMessageHandler` instead of to the `outputChannel` of the `WebSocketInboundChannelAdapter`.

If `useBroker = false` and the received message is of the `SimpMessageType.CONNECT` type, the `WebSocketInboundChannelAdapter` immediately sends a `SimpMessageType.CONNECT_ACK` message to the `WebSocketSession` without sending it to the channel.

Spring's WebSocket Support allows the configuration of only one broker relay. Consequently, we do not require an `AbstractBrokerMessageHandler` reference. It is detected in the Application Context.

For more configuration options, see WebSockets Namespace Support.

38.3. WebSocket Outbound Channel Adapter

The `WebSocketOutboundChannelAdapter`:

1. Accepts Spring Integration messages from its `MessageChannel`
2. Determines the `WebSocketSession id` from the `MessageHeaders`
3. Retrieves the `WebSocketSession` from the provided `IntegrationWebSocketContainer`
4. Delegates the conversion and sending of `WebSocketMessage` work to the appropriate `SubProtocolHandler` from the provided `SubProtocolHandlerRegistry`.

On the client side, the `WebSocketSession id` message header is not required, because `ClientWebSocketContainer` deals only with a single connection and its `WebSocketSession` respectively.

To use the STOMP sub-protocol, you should configure this adapter with a `StompSubProtocolHandler`. Then you can send any STOMP message type to this adapter, using `StompHeaderAccessor.create(StompCommand…)` and a `MessageBuilder`, or just using a `HeaderEnricher` (see Header Enricher).

The rest of this chapter covers largely additional configuration options.

38.4. WebSockets Namespace Support

The Spring Integration WebSocket namespace includes several components described in the remainder of this chapter. To include it in your configuration, use the following namespace declaration in your application context configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-websocket="
http://www.springframework.org/schema/integration/websocket"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    https://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/websocket
    https://www.springframework.org/schema/integration/websocket/spring-
integration-websocket.xsd">
    ...
</beans>
```

38.4.1. `<int-websocket:client-container>` Attributes

The following listing shows the attributes available for the `<int-websocket:client-container>` element:

```
<int-websocket:client-container
                  id=""                          ①
                  client=""                      ②
                  uri=""                         ③
                  uri-variables=""               ④
                  origin=""                      ⑤
                  send-time-limit=""             ⑥
                  send-buffer-size-limit=""      ⑦
                  auto-startup=""                ⑧
                  phase="">                      ⑨
                <int-websocket:http-headers>
                  <entry key="" value=""/>
                </int-websocket:http-headers>    ⑩
</int-websocket:client-container>
```

① The component bean name.

② The `WebSocketClient` bean reference.

③ The `uri` or `uriTemplate` to the target WebSocket service. If you use it as a `uriTemplate` with URI variable placeholders, the `uri-variables` attribute is required.

④ Comma-separated values for the URI variable placeholders within the `uri` attribute value. The values are replaced into the placeholders according to their order in the `uri`. See `UriComponents.expand(Object…uriVariableValues)`.

⑤ The `Origin` Handshake HTTP header value.

⑥ The WebSocket session 'send' timeout limit. Defaults to `10000`.

⑦ The WebSocket session 'send' message size limit. Defaults to `524288`.

⑧ Boolean value indicating whether this endpoint should start automatically. Defaults to `false`, assuming that this container is started from the WebSocket inbound adapter.

⑨ The lifecycle phase within which this endpoint should start and stop. The lower the value, the earlier this endpoint starts and the later it stops. The default is `Integer.MAX_VALUE`. Values can be negative. See `SmartLifeCycle`.

⑩ A `Map` of `HttpHeaders` to be used with the Handshake request.

38.4.2. `<int-websocket:server-container>` Attributes

The following listing shows the attributes available for the `<int-websocket:server-container>` element:

```
<int-websocket:server-container
          id=""                          ①
          path=""                        ②
          handshake-handler=""           ③
          handshake-interceptors=""      ④
          decorator-factories=""         ⑤
          send-time-limit=""             ⑥
          send-buffer-size-limit=""      ⑦
          allowed-origins="">            ⑧
          <int-websocket:sockjs
            client-library-url=""        ⑨
            stream-bytes-limit=""        ⑩
            session-cookie-needed=""     ⑪
            heartbeat-time=""            ⑫
            disconnect-delay=""          ⑬
            message-cache-size=""        ⑭
            websocket-enabled=""         ⑮
            scheduler=""                 ⑯
            message-codec=""             ⑰
            transport-handlers=""        ⑱
            suppress-cors="true"="" />   ⑲
</int-websocket:server-container>
```

① The component bean name.

② A path (or comma-separated paths) that maps a particular request to a `WebSocketHandler`. Supports exact path mapping URIs (such as `/myPath`) and ant-style path patterns (such as `/myPath/**`).

③ The `HandshakeHandler` bean reference. Defaults to `DefaultHandshakeHandler`.

④ List of `HandshakeInterceptor` bean references.

⑤ List of one or more factories (`WebSocketHandlerDecoratorFactory`) that decorate the handler used to process WebSocket messages. This may be useful for some advanced use cases (for example, to allow Spring Security to forcibly close the WebSocket session when the corresponding HTTP session expires). See the Spring Session Project for more information.

⑥ See the same option on the `<int-websocket:client-container>`.

⑦ See the same option on the `<int-websocket:client-container>`.

⑧ The allowed origin header values. You can specify multiple origins as a comma-separated list. This check is mostly designed for browser clients. There is nothing preventing other types of client from modifying the origin header value. When SockJS is enabled and allowed origins are restricted, transport types that do not use origin headers for cross-origin requests (`jsonp-polling`, `iframe-xhr-polling`, `iframe-eventsource`, and `iframe-htmlfile`) are disabled. As a consequence, IE6 and IE7 are not supported, and IE8 and IE9 are supported only without cookies. By default, all origins are allowed.

⑨ Transports with no native cross-domain communication (such as `eventsource` and `htmlfile`) must get a simple page from the “foreign” domain in an invisible iframe so that code in the

iframe can run from a domain local to the SockJS server. Since the iframe needs to load the SockJS javascript client library, this property lets you specify the location from which to load it. By default, it points to `d1fxtkz8shb9d2.cloudfront.net/sockjs-0.3.4.min.js`. However, you can also set it to point to a URL served by the application. Note that it is possible to specify a relative URL, in which case the URL must be relative to the iframe URL. For example, assuming a SockJS endpoint mapped to `/sockjs` and the resulting iframe URL is `/sockjs/iframe.html`, the relative URL must start with "../../" to traverse up to the location above the SockJS mapping. For prefix-based servlet mapping, you may need one more traversal.

⑩ Minimum number of bytes that can be sent over a single HTTP streaming request before it is closed. Defaults to `128K` (that is, 128*1024 or 131072 bytes).

⑪ The `cookie_needed` value in the response from the SockJs `/info` endpoint. This property indicates whether a `JSESSIONID` cookie is required for the application to function correctly (for example, for load balancing or in Java Servlet containers for the use of an HTTP session).

⑫ The amount of time (in milliseconds) when the server has not sent any messages and after which the server should send a heartbeat frame to the client in order to keep the connection from breaking. The default value is `25,000` (25 seconds).

⑬ The amount of time (in milliseconds) before a client is considered disconnected after not having a receiving connection (that is, an active connection over which the server can send data to the client). The default value is `5000`.

⑭ The number of server-to-client messages that a session can cache while waiting for the next HTTP polling request from the client. The default size is `100`.

⑮ Some load balancers do not support WebSockets. Set this option to `false` to disable the WebSocket transport on the server side. The default value is `true`.

⑯ The `TaskScheduler` bean reference. A new `ThreadPoolTaskScheduler` instance is created if no value is provided. This scheduler instance is used for scheduling heart-beat messages.

⑰ The `SockJsMessageCodec` bean reference to use for encoding and decoding SockJS messages. By default, `Jackson2SockJsMessageCodec` is used, which requires the Jackson library to be present on the classpath.

⑱ List of `TransportHandler` bean references.

⑲ Whether to disable automatic addition of CORS headers for SockJS requests. The default value is `false`.

38.4.3. `<int-websocket:outbound-channel-adapter>` Attributes

The following listing shows the attributes available for the `<int-websocket:outbound-channel-adapter>` element:

```
<int-websocket:outbound-channel-adapter
                          id=""                                ①
                          channel=""                           ②
                          container=""                         ③
                          default-protocol-handler=""          ④
                          protocol-handlers=""                 ⑤
                          message-converters=""                ⑥
                          merge-with-default-converters=""     ⑦
                          auto-startup=""                      ⑧
                          phase=""/>                           ⑨
```

① The component bean name. If you do not provide the `channel` attribute, a `DirectChannel` is created and registered in the application context with this `id` attribute as the bean name. In this case, the endpoint is registered with the bean name `id` plus `.adapter`. And the `MessageHandler` is registered with the bean alias `id` plus `.handler`.

② Identifies the channel attached to this adapter.

③ The reference to the `IntegrationWebSocketContainer` bean, which encapsulates the low-level connection and `WebSocketSession` handling operations. Required.

④ Optional reference to a `SubProtocolHandler` instance. It is used when the client did not request a sub-protocol or it is a single protocol-handler. If this reference or a `protocol-handlers` list is not provided, the `PassThruSubProtocolHandler` is used by default.

⑤ List of `SubProtocolHandler` bean references for this channel adapter. If you provide only a single bean reference and do not provide a `default-protocol-handler`, that single `SubProtocolHandler` is used as the `default-protocol-handler`. If you do not set this attribute or `default-protocol-handler`, the `PassThruSubProtocolHandler` is used by default.

⑥ List of `MessageConverter` bean references for this channel adapter.

⑦ Boolean value indicating whether the default converters should be registered after any custom converters. This flag is used only if `message-converters` is provided. Otherwise, all default converters are registered. Defaults to `false`. The default converters are (in order): `StringMessageConverter`, `ByteArrayMessageConverter`, and `MappingJackson2MessageConverter` (if the Jackson library is present on the classpath).

⑧ Boolean value indicating whether this endpoint should start automatically. Defaults to `true`.

⑨ The lifecycle phase within which this endpoint should start and stop. The lower the value, the earlier this endpoint starts and the later it stops. The default is `Integer.MIN_VALUE`. Values can be negative. See `SmartLifeCycle`.

38.4.4. `<int-websocket:inbound-channel-adapter>` Attributes

The following listing shows the attributes available for the `<int-websocket:outbound-channel-adapter>` element:

```
<int-websocket:inbound-channel-adapter
                              id=""  ①
                              channel=""  ②
                              error-channel=""  ③
                              container=""  ④
                              default-protocol-handler=""  ⑤
                              protocol-handlers=""  ⑥
                              message-converters=""  ⑦
                              merge-with-default-converters=""  ⑧
                              send-timeout=""  ⑨
                              payload-type=""  ⑩
                              use-broker=""  ⑪
                              auto-startup=""  ⑫
                              phase=""/>  ⑬
```

① The component bean name. If you do not set the `channel` attribute, a `DirectChannel` is created and registered in the application context with this `id` attribute as the bean name. In this case, the endpoint is registered with the bean name `id` plus `.adapter`.

② Identifies the channel attached to this adapter.

③ The `MessageChannel` bean reference to which the `ErrorMessage` instances should be sent.

④ See the same option on the `<int-websocket:outbound-channel-adapter>`.

⑤ See the same option on the `<int-websocket:outbound-channel-adapter>`.

⑥ See the same option on the `<int-websocket:outbound-channel-adapter>`.

⑦ See the same option on the `<int-websocket:outbound-channel-adapter>`.

⑧ See the same option on the `<int-websocket:outbound-channel-adapter>`.

⑨ Maximum amount of time (in milliseconds) to wait when sending a message to the channel if the channel can block. For example, a `QueueChannel` can block until space is available if its maximum capacity has been reached.

⑩ Fully qualified name of the Java type for the target `payload` to convert from the incoming `WebSocketMessage`. Defaults to `java.lang.String`.

⑪ Indicates whether this adapter sends `non-MESSAGE WebSocketMessage` instances and messages with broker destinations to the `AbstractBrokerMessageHandler` from the application context. When this attribute is `true`, the `Broker Relay` configuration is required. This attribute is used only on the server side. On the client side, it is ignored. Defaults to `false`.

⑫ See the same option on the `<int-websocket:outbound-channel-adapter>`.

⑬ See the same option on the `<int-websocket:outbound-channel-adapter>`.

38.5. Using `ClientStompEncoder`

Starting with version 4.3.13, Spring Integration provides `ClientStompEncoder` (as an extension of the standard `StompEncoder`) for use on the client side of WebSocket channel adapters. For proper client

side message preparation, you must inject an instance of the `ClientStompEncoder` into the `StompSubProtocolHandler`. One problem with the default `StompSubProtocolHandler` is that it was designed for the server side, so it updates the `SEND` `stompCommand` header into `MESSAGE` (as required by the STOMP protocol for the server side). If the client does not send its messages in the proper `SEND` web socket frame, some STOMP brokers do not accept them. The purpose of the `ClientStompEncoder`, in this case, is to override the `stompCommand` header and set it to the `SEND` value before encoding the message to the `byte[]`.

Chapter 39. Web Services Support

This chapter describes Spring Integration's support for web services, including:

- Outbound Web Service Gateways
- Inbound Web Service Gateways
- Web Service Namespace Support
- Outbound URI Configuration
- WS Message Headers
- MTOM Support

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-ws</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-ws:5.3.0.RC1"
```

39.1. Outbound Web Service Gateways

To invoke a web service when you send a message to a channel, you have two options, both of which build upon the Spring Web Services project: `SimpleWebServiceOutboundGateway` and `MarshallingWebServiceOutboundGateway`. The former accepts either a `String` or `javax.xml.transform.Source` as the message payload. The latter supports any implementation of the `Marshaller` and `Unmarshaller` interfaces. Both require a Spring Web Services `DestinationProvider`, to determine the URI of the web service to be called. The following example shows both options for invoking a web service:

```
 simpleGateway = new SimpleWebServiceOutboundGateway(destinationProvider);

 marshallingGateway = new MarshallingWebServiceOutboundGateway(
destinationProvider, marshaller);
```

When using the namespace support (described later), you need only set a URI. Internally, the parser configures a fixed URI `DestinationProvider` implementation. If you need dynamic resolution of the URI at runtime, however, then the `DestinationProvider` can provide such behavior as looking up the URI from a registry. See the Spring Web Services `DestinationProvider` Javadoc for more information about this strategy.

Starting with version 5.0, you can supply the `SimpleWebServiceOutboundGateway` and `MarshallingWebServiceOutboundGateway` with an external `WebServiceTemplate` instance, which you can configure for any custom properties, including `checkConnectionForFault` (which allows your application to deal with non-conforming services).

For more detail on the inner workings, see the Spring Web Services reference guide's chapter covering client access and the chapter covering Object/XML mapping.

39.2. Inbound Web Service Gateways

To send a message to a channel upon receiving a web service invocation, you again have two options: `SimpleWebServiceInboundGateway` and `MarshallingWebServiceInboundGateway`. The former extracts a `javax.xml.transform.Source` from the `WebServiceMessage` and sets it as the message payload. The latter supports implementation of the `Marshaller` and `Unmarshaller` interfaces. If the incoming web service message is a SOAP message, the SOAP action header is added to the headers of the `Message` that is forwarded onto the request channel. The following example shows both options:

```
simpleGateway = new SimpleWebServiceInboundGateway();
simpleGateway.setRequestChannel(forwardOntoThisChannel);
simpleGateway.setReplyChannel(listenForResponseHere); //Optional

marshallingGateway = new MarshallingWebServiceInboundGateway(marshaller);
//set request and optionally reply channel
```

Both gateways implement the Spring Web Services `MessageEndpoint` interface, so they can be configured with a `MessageDispatcherServlet` as per standard Spring Web Services configuration.

For more detail on how to use these components, see the Spring Web Services reference guide's chapter covering creating a web service. The chapter covering Object/XML mapping is also applicable again.

To add the `SimpleWebServiceInboundGateway` and `MarshallingWebServiceInboundGateway` configurations to the Spring WS infrastructure, you should add the `EndpointMapping` definition between `MessageDispatcherServlet` and the target `MessageEndpoint` implementations, as you would for a normal Spring WS application. For this purpose (from the Spring Integration perspective), Spring WS provides the following convenient `EndpointMapping` implementations:

- `o.s.ws.server.endpoint.mapping.UriEndpointMapping`

- o.s.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping
- o.s.ws.soap.server.endpoint.mapping.SoapActionEndpointMapping
- o.s.ws.server.endpoint.mapping.XPathPayloadEndpointMapping

You must specify the beans for these classes in the application context and reference the SimpleWebServiceInboundGateway and/or MarshallingWebServiceInboundGateway bean definitions according to the WS mapping algorithm.

See the endpoint mappings for more information.

39.3. Web Service Namespace Support

To configure an outbound web service gateway, use the outbound-gateway element from the ws namespace, as the following example shows:

```
<int-ws:outbound-gateway id="simpleGateway"
                         request-channel="inputChannel"
                         uri="https://example.org"/>
```

This example does not provide a 'reply-channel'. If the web service returns a non-empty response, the Message containing that response is sent to the reply channel defined in the request message's REPLY_CHANNEL header. If that is not available, a channel resolution exception is thrown. If you want to send the reply to another channel instead, provide a 'reply-channel' attribute on the 'outbound-gateway' element.

By default, when you invoke a web service that returns an empty response after using a String payload for the request Message, no reply Message is sent. Therefore, you need not set a 'reply-channel' or have a REPLY_CHANNEL header in the request Message. If you actually do want to receive the empty response as a Message, you can set the 'ignore-empty-responses' attribute to false. Doing so works only for String objects, because using a Source or a Document object leads to a null response and consequently never generates a reply Message.

To set up an inbound Web Service Gateway, use the inbound-gateway element, as the following example shows:

```
<int-ws:inbound-gateway id="simpleGateway"
                        request-channel="inputChannel"/>
```

To use Spring OXM marshallers or unmarshallers, you must provide bean references. The following example shows how to provide a bean reference for an outbound marshalling gateway:

```
<int-ws:outbound-gateway id="marshallingGateway"
                         request-channel="requestChannel"
                         uri="https://example.org"
                         marshaller="someMarshaller"
                         unmarshaller="someUnmarshaller"/>
```

The following example shows how to provide a bean reference for an inbound marshalling gateway:

```
<int-ws:inbound-gateway id="marshallingGateway"
                        request-channel="requestChannel"
                        marshaller="someMarshaller"
                        unmarshaller="someUnmarshaller"/>
```

Most `Marshaller` implementations also implement the `Unmarshaller` interface. When using such a `Marshaller`, only the `marshaller` attribute is necessary. Even when using a `Marshaller`, you may also provide a reference for the `request-callback` on the outbound gateways.

For either outbound gateway type, you can specify a `destination-provider` attribute instead of the `uri` (exactly one of them is required). You can then reference any Spring Web Services `DestinationProvider` implementation (for example, to lookup the URI from a registry at runtime).

For either outbound gateway type, the `message-factory` attribute can also be configured with a reference to any Spring Web Services `WebServiceMessageFactory` implementation.

For the simple inbound gateway type, you can set the `extract-payload` attribute to `false` to forward the entire `WebServiceMessage` instead of just its payload as a `Message` to the request channel. Doing so might be useful, for example, when a custom transformer works against the `WebServiceMessage` directly.

Starting with version 5.0, the `web-service-template` reference attribute lets you inject a `WebServiceTemplate` with any possible custom properties.

39.4. Web Service Java DSL Support

The equivalent configuration for the gateways shown in Web Service Namespace Support are shown in the following snippets:

```
@Bean
IntegrationFlow inbound() {
    return IntegrationFlows.from(Ws.simpleInboundGateway()
                .id("simpleGateway"))
        ...
        .get();
}
```

```
@Bean
IntegrationFlow outboundMarshalled() {
    return f -> f.handle(Ws.marshallingOutboundGateway()
                    .id("marshallingGateway")
                    .marshaller(someMarshaller())
                    .unmarshaller(someUnmarshallller()))
        ...
}
```

```
@Bean
IntegrationFlow inboundMarshalled() {
    return IntegrationFlows.from(Ws.marshallingInboundGateway()
                .marshaller(someMarshaller())
                .unmarshaller(someUnmarshallller())
                .id("marshallingGateway"))
        ...
        .get();
}
```

Other properties can be set on the endpoint specs in a fluent manner (with the properties depending on whether or not an external `WebServiceTemplate` has been provided for outbound gateways). Examples:

```
.from(Ws.simpleInboundGateway()
                .extractPayload(false))
```

```
.handle(Ws.simpleOutboundGateway(template)
            .uri(uri)
            .sourceExtractor(sourceExtractor)
            .encodingMode(DefaultUriBuilderFactory.EncodingMode.NONE)
            .headerMapper(headerMapper)
            .ignoreEmptyResponses(true)
            .requestCallback(requestCallback)
            .uriVariableExpressions(uriVariableExpressions)
            .extractPayload(false))
)
```

```
.handle(Ws.marshallingOutboundGateway()
            .destinationProvider(destinationProvider)
            .marshaller(marshaller)
            .unmarshaller(unmarshaller)
            .messageFactory(messageFactory)
            .encodingMode(DefaultUriBuilderFactory.EncodingMode.VALUES_ONLY)
            .faultMessageResolver(faultMessageResolver)
            .headerMapper(headerMapper)
            .ignoreEmptyResponses(true)
            .interceptors(interceptor)
            .messageSenders(messageSender)
            .requestCallback(requestCallback)
            .uriVariableExpressions(uriVariableExpressions))
```

```
.handle(Ws.marshallingOutboundGateway(template)
            .uri(uri)
            .encodingMode(DefaultUriBuilderFactory.EncodingMode.URI_COMPONENT)
            .headerMapper(headerMapper)
            .ignoreEmptyResponses(true)
            .requestCallback(requestCallback)
            .uriVariableExpressions(uriVariableExpressions))
)
```

39.5. Outbound URI Configuration

For all URI schemes supported by Spring Web Services (see URIs and Transports) `<uri-variable/>` substitution is provided. The following example shows how to define it:

```
<ws:outbound-gateway id="gateway" request-channel="input"
        uri="https://springsource.org/{thing1}-{thing2}">
    <ws:uri-variable name="thing1" expression="payload.substring(1,7)"/>
    <ws:uri-variable name="thing2" expression="headers.x"/>
</ws:outbound-gateway>

<ws:outbound-gateway request-channel="inputJms"
        uri="jms:{destination}?deliveryMode={deliveryMode}&amp;priority={priority}"
        message-sender="jmsMessageSender">
    <ws:uri-variable name="destination" expression="headers.jmsQueue"/>
    <ws:uri-variable name="deliveryMode" expression="headers.deliveryMode"/>
    <ws:uri-variable name="priority" expression="headers.jms_priority"/>
</ws:outbound-gateway>
```

If you supply a `DestinationProvider`, variable substitution is not supported and a configuration error occurs if you provide variables.

39.5.1. Controlling URI Encoding

By default, the URL string is encoded (see `UriComponentsBuilder`) to the URI object before sending the request. In some scenarios with a non-standard URI, it is undesirable to perform the encoding. The `<ws:outbound-gateway/>` element provides an `encoding-mode` attribute. To disable encoding the URL, set this attribute to `NONE` (by default, it is `TEMPLATE_AND_VALUES`). If you wish to partially encode some of the URL, you can do so by using an `expression` within a `<uri-variable/>`, as the following example shows:

```
<ws:outbound-gateway url="https://somehost/%2f/fooApps?bar={param}" encoding-mode
="NONE">
          <http:uri-variable name="param"
            expression="T(org.apache.commons.httpclient.util.URIUtil)
                                          .encodeWithinQuery('Hello World!')"/>
</ws:outbound-gateway>
```

If you set `DestinationProvider`, `encoding-mode` is ignored.

39.6. WS Message Headers

The Spring Integration web service gateways automatically map the SOAP action header. By default, it is copied to and from Spring Integration `MessageHeaders` by using the `DefaultSoapHeaderMapper`.

You can pass in your own implementation of SOAP-specific header mappers, as the gateways have properties to support doing so.

Unless explicitly specified by the `requestHeaderNames` or `replyHeaderNames` properties of the

DefaultSoapHeaderMapper, any user-defined SOAP headers are not copied to or from a SOAP Message.

When you use the XML namespace for configuration, you can set these properties by using the mapped-request-headers and mapped-reply-headers attributes, you can provide a custom mapper by setting the header-mapper attribute.

When mapping user-defined headers, the values can also contain simple wildcard patterns (such myheader* or **myheader**). For example, if you need to copy all user-defined headers, you can use the wildcard character: .

Starting with version 4.1, the AbstractHeaderMapper (a DefaultSoapHeaderMapper superclass) lets the NON_STANDARD_HEADERS token be configured for the requestHeaderNames and replyHeaderNames properties (in addition to existing STANDARD_REQUEST_HEADERS and STANDARD_REPLY_HEADERS) to map all user-defined headers.

Rather than using the wildcard (*), we recommend using the following combination : STANDARD_REPLY_HEADERS, NON_STANDARD_HEADERS. Doing so avoids mapping request headers to the reply.

Starting with version 4.3, you can negate patterns in the header mappings by preceding the pattern with !. Negated patterns get priority, so a list such as STANDARD_REQUEST_HEADERS,thing1,thing*,!thing2,!thing3,qux,!thing1 does not map thing1, thing2, or thing3. It does map the standard headers, thing4, and qux. (Note that thing1 is included in both non-negated and negated forms. Because negated values take precedence, thing1 is not mapped.)

If you have a user-defined header that begins with ! that you do wish to map, you can escape it with \, as follows: STANDARD_REQUEST_HEADERS,\!myBangHeader. A !myBangHeader is then mapped.

Inbound SOAP headers (request headers for the inbound gateway and reply headers for the outbound gateway) are mapped as SoapHeaderElement objects. You can explore the contents by accessing the Source:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <soapenv:Header>
        <auth>
            <username>user</username>
            <password>pass</password>
        </auth>
        <bar>BAR</bar>
        <baz>BAZ</baz>
        <qux>qux</qux>
    </soapenv:Header>
    <soapenv:Body>
        ...
    </soapenv:Body>
</soapenv:Envelope>
```

If `mapped-request-headers` is `auth, ca*`, the `auth`, `cat`, and `can` headers are mapped, but `qux` is not mapped.

The following example shows how to get a value named `user` from a header named `auth`:

```
...
SoapHeaderElement header = (SoapHeaderElement) headers.get("auth");
DOMSource source = (DOMSource) header.getSource();
NodeList nodeList = source.getNode().getChildNodes();
assertEquals("username", nodeList.item(0).getNodeName());
assertEquals("user", nodeList.item(0).getFirstChild().getNodeValue());
...
```

Starting with version 5.0, the `DefaultSoapHeaderMapper` supports user-defined headers of type `javax.xml.transform.Source` and populates them as child nodes of the `<soapenv:Header>`. The following example shows how to do so:

```
Map<String, Object> headers = new HashMap<>();

String authXml =
     "<auth xmlns='http://test.auth.org'>"
           + "<username>user</username>"
           + "<password>pass</password>"
           + "</auth>";
headers.put("auth", new StringSource(authXml));
...
DefaultSoapHeaderMapper mapper = new DefaultSoapHeaderMapper();
mapper.setRequestHeaderNames("auth");
```

The result of the preceding examples is the following SOAP envelope:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <soapenv:Header>
        <auth xmlns="http://test.auth.org">
            <username>user</username>
            <password>pass</password>
        </auth>
    </soapenv:Header>
    <soapenv:Body>
        ...
    </soapenv:Body>
</soapenv:Envelope>
```

39.7. MTOM Support

The marshalling inbound and outbound web service gateways support attachments directly through built-in functionality of the marshaller (for example, `Jaxb2Marshaller` provides the `mtomEnabled` option). Starting with version 5.0, the simple web service gateways can directly operate with inbound and outbound `MimeMessage` instances, which have an API to manipulate attachments. When you need to send web service message with attachments (either a reply from a server or a client request) you should use the `WebServiceMessageFactory` directly and send a `WebServiceMessage` with attachments as a `payload` to the request or reply channel of the gateway. The following example shows how to do so:

```
WebServiceMessageFactory messageFactory = new SaajSoapMessageFactory
(MessageFactory.newInstance());
MimeMessage webServiceMessage = (MimeMessage) messageFactory
.createWebServiceMessage();

String request = "<test>foo</test>";

TransformerFactory transformerFactory = TransformerFactory.newInstance();
Transformer transformer = transformerFactory.newTransformer();
transformer.transform(new StringSource(request), webServiceMessage
.getPayloadResult());

webServiceMessage.addAttachment("myAttachment", new ByteArrayResource("my_data"
.getBytes()), "plain/text");

this.webServiceChannel.send(new GenericMessage<>(webServiceMessage));
```

Chapter 40. XML Support - Dealing with XML Payloads

Spring Integration's XML support extends the core of Spring Integration with the following components:

- Marshalling Transformer
- Unmarshalling Transformer
- XSLT Transformer
- XPath Transformer
- XPath Splitter
- XPath Router
- XPath Header Enricher
- XPath Filter
- #xpath SpEL Function
- Validating Filter

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-xml</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-xml:5.3.0.RC1"
```

These components make working with XML messages in Spring Integration simpler. The messaging components work with XML that is represented in a range of formats, including instances of `java.lang.String`, `org.w3c.dom.Document`, and `javax.xml.transform.Source`. However, where a DOM representation is required (for example, in order to evaluate an XPath expression), the `String` payload is converted into the required type and then converted back to `String`. Components that require an instance of `DocumentBuilder` create a namespace-aware instance if you do not provide one. When you require greater control over document creation, you can provide an appropriately configured instance of `DocumentBuilder`.

40.1. Namespace Support

All components within the Spring Integration XML module provide namespace support. In order to enable namespace support, you need to import the schema for the Spring Integration XML Module. The following example shows a typical setup:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-xml="http://www.springframework.org/schema/integration/xml"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    https://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/xml
    https://www.springframework.org/schema/integration/xml/spring-integration-
xml.xsd">
</beans>
```

40.1.1. XPath Expressions

Many of the components within the Spring Integration XML module work with XPath Expressions. Each of those components either references an XPath Expression that has been defined as a top-level element or uses a nested `<xpath-expression/>` element.

All forms of XPath expressions result in the creation of an `XPathExpression` that uses the Spring `org.springframework.xml.xpath.XPathExpressionFactory`. When XPath expressions are created, the best XPath implementation that is available on the classpath is used (either JAXP 1.3+ or Jaxen, with JAXP being preferred).

Internally, Spring Integration uses the XPath functionality provided by the Spring Web Services project (www.spring.io/spring-ws). Specifically, we use the Spring Web Services XML module (spring-xml-x.x.x.jar). For a deeper understanding, see the respective documentation at docs.spring.io/spring-ws/docs/current/reference/#xpath.

Here is an overview of all available configuration parameters of the `xpath-expression` element: The following listing shows the available attributes for the `xpath-expression` element:

```
<int-xml:xpath-expression expression="" ①
          id=""                         ②
          namespace-map=""              ③
          ns-prefix=""                  ④
          ns-uri="">                    ⑤
    <map></map>                         ⑥
</int-xml:xpath-expression>
```

① Defines an XPath expression. Required.

② The identifier of the underlying bean definition. It is an instance of `org.springframework.xml.xpath.XPathExpression`. Optional.

③ Reference to a map that contains namespaces. The key of the map defines the namespace prefix, and the value of the map sets the namespace URI. It is not valid to specify both this attribute and the `map` element or the `ns-prefix` and `ns-uri` attributes. Optional.

④ Lets you set the namespace prefix directly as an attribute on the XPath expression element. If you set `ns-prefix`, you must also set the `ns-uri` attribute. Optional.

⑤ Lets you directly set the namespace URI as an attribute on the XPath expression element. If you set `ns-uri`, you must also set the `ns-prefix` attribute. Optional.

⑥ Defines a map that contains namespaces. Only one `map` child element is allowed. The key of the map defines the namespace prefix, and the value of the map sets the namespace URI. It is not valid to specify both this element and the `map` attribute or set the `ns-prefix` and `ns-uri` attributes. Optional.

Providing Namespaces (Optional) to XPath Expressions

For the XPath Expression Element, you can provide namespace information as configuration parameters. You can define namespaces by using one of the following choices:

- Reference a map by using the `namespace-map` attribute
- Provide a map of namespaces by using the `map` sub-element
- Specify the `ns-prefix` and `ns-uri` attributes

All three options are mutually exclusive. Only one option can be set.

The following example shows several different ways to use XPath expressions, including the options for setting the XML namespaces mentioned earlier:

```
<int-xml:xpath-filter id="filterReferencingXPathExpression"
                      xpath-expression-ref="refToXpathExpression"/>

<int-xml:xpath-expression id="refToXpathExpression" expression="/name"/>

<int-xml:xpath-filter id="filterWithoutNamespace">
    <int-xml:xpath-expression expression="/name"/>
</int-xml:xpath-filter>

<int-xml:xpath-filter id="filterWithOneNamespace">
    <int-xml:xpath-expression expression="/ns1:name"
                              ns-prefix="ns1" ns-uri="www.example.org"/>
</int-xml:xpath-filter>

<int-xml:xpath-filter id="filterWithTwoNamespaces">
    <int-xml:xpath-expression expression="/ns1:name/ns2:type">
        <map>
            <entry key="ns1" value="www.example.org/one"/>
            <entry key="ns2" value="www.example.org/two"/>
        </map>
    </int-xml:xpath-expression>
</int-xml:xpath-filter>

<int-xml:xpath-filter id="filterWithNamespaceMapReference">
    <int-xml:xpath-expression expression="/ns1:name/ns2:type"
                              namespace-map="defaultNamespaces"/>
</int-xml:xpath-filter>

<util:map id="defaultNamespaces">
    <util:entry key="ns1" value="www.example.org/one"/>
    <util:entry key="ns2" value="www.example.org/two"/>
</util:map>
```

Using XPath Expressions with Default Namespaces

When working with default namespaces, you may run into situations that behave differently than you might expect. Assume we have the following XML document (which represents an order of two books):

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
    <orderItem>
        <isbn>0321200683</isbn>
        <quantity>2</quantity>
    </orderItem>
    <orderItem>
        <isbn>1590596439</isbn>
        <quantity>1</quantity>
    </orderItem>
</order>
```

This document does not declare a namespace. Therefore, applying the following XPath Expression works as expected:

```
<int-xml:xpath-expression expression="/order/orderItem" />
```

You might expect that the same expression also works for the following XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<order xmlns="http://www.example.org/orders">
    <orderItem>
        <isbn>0321200683</isbn>
        <quantity>2</quantity>
    </orderItem>
    <orderItem>
        <isbn>1590596439</isbn>
        <quantity>1</quantity>
    </orderItem>
</order>
```

The preceding example looks exactly the same as the previous example but declares a default namespace.

However, the previous XPath expression (`/order/orderItem`) fails in this case.

In order to solve this issue, you must provide a namespace prefix and a namespace URI either by setting the `ns-prefix` and `ns-uri` attributes or by setting the `namespace-map` attribute. The namespace URI must match the namespace declared in your XML document. In the preceding example, that is `www.example.org/orders`.

You can, however, arbitrarily choose the namespace prefix. In fact, providing an empty string

actually works. (However, null is not allowed.) In the case of a namespace prefix consisting of an empty string, your Xpath expression must use a colon (":") to indicate the default namespace. If you leave off the colon, the XPath expression does not match. The following XPath Expression matches against the XML document in the preceding example:

```
<int-xml:xpath-expression expression="/:order/:orderItem"
    ns-prefix="" ns-uri="https://www.example.org/prodcuts"/>
```

You can also provide any other arbitrarily chosen namespace prefix. The following XPath expression (which use the `myorder` namespace prefix) also matches:

```
<int-xml:xpath-expression expression="/myorder:order/myorder:orderItem"
    ns-prefix="myorder" ns-uri="https://www.example.org/prodcuts"/>
```

The namespace URI is the really important piece of information, not the prefix. Thehttps://github.com/jaxen-xpath/jaxen[Jaxen] summarizes the point very well:

> In XPath 1.0, all unprefixed names are unqualified. There is no requirement that the prefixes used in the XPath expression are the same as the prefixes used in the document being queried. Only the namespace URIs need to match, not the prefixes.

40.2. Transforming XML Payloads

This section covers how to transform XML payloads

40.2.1. Configuring Transformers as Beans

This section will explain the workings of the following transformers and how to configure them as beans:

- UnmarshallingTransformer
- MarshallingTransformer
- XsltPayloadTransformer

All of the XML transformers extend either `AbstractTransformer` or `AbstractPayloadTransformer` and therefore implement `Transformer`. When configuring XML transformers as beans in Spring Integration, you would normally configure the `Transformer` in conjunction with a `MessageTransformingHandler`. This lets the transformer be used as an endpoint. Finally, we discuss the namespace support , which allows for configuring the transformers as elements in XML.

UnmarshallingTransformer

An `UnmarshallingTransformer` lets an XML `Source` be unmarshalled by using implementations of the Spring OXM `Unmarshaller`. Spring's Object/XML Mapping support provides several implementations that support marshalling and unmarshalling by using JAXB, Castor, JiBX, and others. The unmarshaller requires an instance of `Source`. If the message payload is not an instance of `Source`, conversion is still attempted. Currently, `String`, `File`, `byte[]` and `org.w3c.dom.Document` payloads are supported. To create a custom conversion to a `Source`, you can inject an implementation of a `SourceFactory`.

If you do not explicitly set a `SourceFactory`, the property on the `UnmarshallingTransformer` is, by default, set to a `DomSourceFactory`.

Starting with version 5.0, the `UnmarshallingTransformer` also supports an `org.springframework.ws.mime.MimeMessage` as the incoming payload. This can be useful when we receive a raw `WebServiceMessage` with MTOM attachments over SOAP . See MTOM Support for more information.

The following example shows how to define an unmarshalling transformer:

```
<bean id="unmarshallingTransformer" class=
"o.s.i.xml.transformer.UnmarshallingTransformer">
    <constructor-arg>
        <bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
            <property name="contextPath" value="org.example" />
        </bean>
    </constructor-arg>
</bean>
```

Using `MarshallingTransformer`

The `MarshallingTransformer` lets an object graph be converted into XML by using a Spring OXM `Marshaller`. By default, the `MarshallingTransformer` returns a `DomResult`. However, you can control the type of result by configuring an alternative `ResultFactory`, such as `StringResultFactory`. In many cases, it is more convenient to transform the payload into an alternative XML format. To do so, configure a `ResultTransformer`. Spring integration provides two implementations, one that converts to `String` and another that converts to `Document`. The following example configures a marshalling transformer that transforms to a document:

```
<bean id="marshallingTransformer" class="o.s.i.xml.transformer.MarshallingTransformer
">
    <constructor-arg>
        <bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
            <property name="contextPath" value="org.example"/>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean class="o.s.i.xml.transformer.ResultToDocumentTransformer"/>
    </constructor-arg>
</bean>
```

By default, the `MarshallingTransformer` passes the payload object to the `Marshaller`. However, if its boolean `extractPayload` property is set to `false`, the entire `Message` instance is passed to the `Marshaller` instead. That may be useful for certain custom implementations of the `Marshaller` interface, but, typically, the payload is the appropriate source object for marshalling when you delegate to any of the various `Marshaller` implementations.

XsltPayloadTransformer

The `XsltPayloadTransformer` transforms XML payloads by using Extensible Stylesheet Language Transformations (XSLT). The transformer's constructor requires an instance of either `Resource` or `Templates` to be passed in. Passing in a `Templates` instance allows for greater configuration of the `TransformerFactory` used to create the template instance.

As with the `UnmarshallingTransformer`, the `XsltPayloadTransformer` does the actual XSLT transformation against instances of `Source`. Therefore, if the message payload is not an instance of `Source`, conversion is still attempted. `String` and `Document` payloads are supported directly.

To create a custom conversion to a `Source`, you can inject an implementation of a `SourceFactory`.

If a `SourceFactory` is not set explicitly, the property on the `XsltPayloadTransformer` is, by default, set to a `DomSourceFactory`.

By default, the `XsltPayloadTransformer` creates a message with a `Result` payload, similar to the `XmlPayloadMarshallingTransformer`. You can customize this by providing a `ResultFactory` or a `ResultTransformer`.

The following example configures a bean that works as an XSLT payload transformer:

```
<bean id="xsltPayloadTransformer" class="o.s.i.xml.transformer.XsltPayloadTransformer
">
  <constructor-arg value="classpath:org/example/xsl/transform.xsl"/>
  <constructor-arg>
    <bean class="o.s.i.xml.transformer.ResultToDocumentTransformer"/>
  </constructor-arg>
</bean>
```

Starting with Spring Integration 3.0, you can specify the transformer factory class name by using a constructor argument. You can do so by using the `transformer-factory-class` attribute when you use the namespace.

Using `ResultTransformer` Implementations

Both the `MarshallingTransformer` and the `XsltPayloadTransformer` let you specify a `ResultTransformer`. Thus, if the marshalling or XSLT transformation returns a `Result`, you have the option to also use a `ResultTransformer` to transform the `Result` into another format. Spring Integration provides two concrete `ResultTransformer` implementations:

- `ResultToDocumentTransformer`
- `ResultToStringTransformer`

By default, the `MarshallingTransformer` always returns a `Result`. By specifying a `ResultTransformer`, you can customize the type of payload returned.

The behavior is slightly more complex for the `XsltPayloadTransformer`. By default, if the input payload is an instance of `String` or `Document` the `resultTransformer` property is ignored.

However, if the input payload is a `Source` or any other type, the `resultTransformer` property is applied. Additionally, you can set the `alwaysUseResultFactory` property to `true`, which also causes the specified `resultTransformer` to be used.

For more information and examples, see Namespace Configuration and Result Transformers.

40.2.2. Namespace Support for XML Transformers

Namespace support for all XML transformers is provided in the Spring Integration XML namespace, a template for which was shown earlier. The namespace support for transformers creates an instance of either `EventDrivenConsumer` or `PollingConsumer`, according to the type of the provided input channel. The namespace support is designed to reduce the amount of XML configuration by allowing the creation of an endpoint and transformer that use one element.

Using an `UnmarshallingTransformer`

The namespace support for the `UnmarshallingTransformer` is shown below. Since the namespace create an endpoint instance rather than a transformer, you can nest a poller within the element to control the polling of the input channel. The following example shows how to do so:

```
<int-xml:unmarshalling-transformer id="defaultUnmarshaller"
    input-channel="input" output-channel="output"
    unmarshaller="unmarshaller"/>

<int-xml:unmarshalling-transformer id="unmarshallerWithPoller"
    input-channel="input" output-channel="output"
    unmarshaller="unmarshaller">
    <int:poller fixed-rate="2000"/>
<int-xml:unmarshalling-transformer/>
```

Using a `MarshallingTransformer`

The namespace support for the marshalling transformer requires an `input-channel`, an `output-channel`, and a reference to a `marshaller`. You can use the optional `result-type` attribute to control the type of result created. Valid values are `StringResult` or `DomResult` (the default). The following example configures a marshalling transformer:

```
<int-xml:marshalling-transformer
     input-channel="marshallingTransformerStringResultFactory"
     output-channel="output"
     marshaller="marshaller"
     result-type="StringResult" />

<int-xml:marshalling-transformer
    input-channel="marshallingTransformerWithResultTransformer"
    output-channel="output"
    marshaller="marshaller"
    result-transformer="resultTransformer" />

<bean id="resultTransformer" class=
"o.s.i.xml.transformer.ResultToStringTransformer"/>
```

Where the provided result types do not suffice, you can provide a reference to a custom implementation of `ResultFactory` as an alternative to setting the `result-type` attribute by using the `result-factory` attribute. The `result-type` and `result-factory` attributes are mutually exclusive.

Internally, the `StringResult` and `DomResult` result types are represented by the `ResultFactory` implementations: `StringResultFactory` and `DomResultFactory` respectively.

Using an `XsltPayloadTransformer`

Namespace support for the `XsltPayloadTransformer` lets you either pass in a `Resource` (in order to create the `Templates` instance) or pass in a pre-created `Templates` instance as a reference. As with the

marshalling transformer, you can control the type of the result output by specifying either the `result-factory` or the `result-type` attribute. When you need to convert result before sending, you can use a `result-transformer` attribute to reference an implementation of `ResultTransformer`.

If you specify the `result-factory` or the `result-type` attribute, the `alwaysUseResultFactory` property on the underlying `XsltPayloadTransformer` is set to `true` by the `XsltPayloadTransformerParser`.

The following example configures two XSLT transformers:

```
<int-xml:xslt-transformer id="xsltTransformerWithResource"
    input-channel="withResourceIn" output-channel="output"
    xsl-resource="org/springframework/integration/xml/config/test.xsl"/>

<int-xml:xslt-transformer id="xsltTransformerWithTemplatesAndResultTransformer"
    input-channel="withTemplatesAndResultTransformerIn" output-channel="output"
    xsl-templates="templates"
    result-transformer="resultTransformer"/>
```

You may need to have access to `Message` data, such as the `Message` headers, in order to assist with transformation. For example, you may need to get access to certain `Message` headers and pass them on as parameters to a transformer (for example, `transformer.setParameter(..)`). Spring Integration provides two convenient ways to accomplish this, as the following example shows:

```
<int-xml:xslt-transformer id="paramHeadersCombo"
    input-channel="paramHeadersComboChannel" output-channel="output"
    xsl-resource="classpath:transformer.xslt"
    xslt-param-headers="testP*, *foo, bar, baz">

    <int-xml:xslt-param name="helloParameter" value="hello"/>
    <int-xml:xslt-param name="firstName" expression="headers.fname"/>
</int-xml:xslt-transformer>
```

If message header names match one-to-one to parameter names, you can use the `xslt-param-headers` attribute. In it, you can use wildcards for simple pattern matching. It supports the following simple pattern styles: `xxx*`, `xxx`, `*xxx`, and `xxx*yyy`.

You can also configure individual XSLT parameters by using the `<xslt-param/>` element. On that element, you can set the `expression` attribute or the `value` attribute. The `expression` attribute should be any valid SpEL expression with the `Message` being the root object of the expression evaluation context. The `value` attribute (as with any `value` in Spring beans) lets you specify simple scalar values. You can also use property placeholders (such as `${some.value}`). So, with the `expression` and `value` attributes, you can map XSLT parameters to any accessible part of the `Message` as well as any literal value.

Starting with Spring Integration 3.0, you can now specify the transformer factory class name by setting the `transformer-factory-class` attribute.

40.2.3. Namespace Configuration and Result Transformers

We cover using result transformers in Using `ResultTransformer` Implementations. The examples in this section use XML namespace configuration to illustrates several special use cases. First, we define the `ResultTransformer`, as the following example shows:

```
<beans:bean id="resultToDoc" class=
"o.s.i.xml.transformer.ResultToDocumentTransformer"/>
```

This `ResultTransformer` accepts either a `StringResult` or a `DOMResult` as input and converts the input into a `Document`.

Now we can declare the transformer, as follows:

```
<int-xml:xslt-transformer input-channel="in" output-channel="fahrenheitChannel"
    xsl-resource="classpath:noop.xslt" result-transformer="resultToDoc"/>
```

If the incoming message's payload is of type `Source`, then, as a first step, the `Result` is determined by using the `ResultFactory`. As we did not specify a `ResultFactory`, the default `DomResultFactory` is used, meaning that the transformation yields a `DomResult`.

However, as we specified a `ResultTransformer`, it is used and the resulting `Message` payload is of type `Document`.

The specified `ResultTransformer` is ignored with `String` or `Document` payloads. If the incoming message's payload is of type `String`, the payload after the XSLT transformation is a `String`. Similarly, if the incoming message's payload is of type `Document`, the payload after the XSLT transformation is a `Document`.

If the message payload is not a `Source`, a `String`, or a `Document`, as a fallback option, we try to create a `Source` by using the default `SourceFactory`. As we did not specify a `SourceFactory` explicitly by using the `source-factory` attribute, the default `DomSourceFactory` is used. If successful, the XSLT transformation is executed as if the payload was of type `Source`, as described in the previous paragraphs.

The `DomSourceFactory` supports the creation of a `DOMSource` from a `Document`, a `File`, or a `String` payload.

The next transformer declaration adds a `result-type` attribute that uses `StringResult` as its value. The `result-type` is internally represented by the `StringResultFactory`. Thus, you could have also added a reference to a `StringResultFactory`, by using the `result-factory` attribute, which would

have been the same. The following example shows that transformer declaration:

```
<int-xml:xslt-transformer input-channel="in" output-channel="fahrenheitChannel"
        xsl-resource="classpath:noop.xslt" result-transformer="resultToDoc"
        result-type="StringResult"/>
```

Because we use a `ResultFactory`, the `alwaysUseResultFactory` property of the `XsltPayloadTransformer` class is implicitly set to `true`. Consequently, the referenced `ResultToDocumentTransformer` is used.

Therefore, if you transform a payload of type `String`, the resulting payload is of type `Document`.

`XsltPayloadTransformer` **and** `<xsl:output method="text"/>`

`<xsl:output method="text"/>` tells the XSLT template to produce only text content from the input source. In this particular case, we have no reason to use a `DomResult`. Therefore, the `XsltPayloadTransformer` defaults to `StringResult` if the output property called `method` of the underlying `javax.xml.transform.Transformer` returns `text`. This coercion is performed independently from the inbound payload type. This behavior is available only you set the if the `result-type` attribute or the `result-factory` attribute for the `<int-xml:xslt-transformer>` component.

40.3. Transforming XML Messages with XPath

When it comes to message transformation, XPath is a great way to transform messages that have XML payloads. You can do so by defining XPath transformers with the `<xpath-transformer/>` element.

40.3.1. Simple XPath Transformation

Consider following transformer configuration:

```
<int-xml:xpath-transformer input-channel="inputChannel" output-channel=
"outputChannel"
      xpath-expression="/person/@name" />
```

Also consider the following `Message`:

```
Message<?> message =
  MessageBuilder.withPayload("<person name='John Doe' age='42' married='true'/>")
.build();
```

After sending this message to the 'inputChannel', the XPath transformer configured earlier

transforms this XML Message to a simple `Message` with a payload of 'John Doe', all based on the simple XPath Expression specified in the `xpath-expression` attribute.

XPath also lets you perform simple conversion of an extracted element to a desired type. Valid return types are defined in `javax.xml.xpath.XPathConstants` and follow the conversion rules specified by the `javax.xml.xpath.XPath` interface.

The following constants are defined by the `XPathConstants` class: `BOOLEAN`, `DOM_OBJECT_MODEL`, `NODE`, `NODESET`, `NUMBER`, and `STRING`.

You can configure the desired type by using the `evaluation-type` attribute of the `<xpath-transformer/>` element, as the following example shows (twice):

```
<int-xml:xpath-transformer input-channel="numberInput" xpath-expression=
"/person/@age"
                           evaluation-type="NUMBER_RESULT" output-channel="output
"/>

<int-xml:xpath-transformer input-channel="booleanInput"
                           xpath-expression="/person/@married = 'true'"
                           evaluation-type="BOOLEAN_RESULT" output-channel="
output"/>
```

40.3.2. Node Mappers

If you need to provide custom mapping for the node extracted by the XPath expression, you can provide a reference to the implementation of the `org.springframework.xml.xpath.NodeMapper` (an interface used by `XPathOperations` implementations for mapping `Node` objects on a per-node basis). To provide a reference to a `NodeMapper`, you can use the `node-mapper` attribute, as the following example shows:

```
<int-xml:xpath-transformer input-channel="nodeMapperInput" xpath-expression=
"/person/@age"
                           node-mapper="testNodeMapper" output-channel="output"/>
```

The following example shows a `NodeMapper` implementation that works with the preceding example:

```
class TestNodeMapper implements NodeMapper {
  public Object mapNode(Node node, int nodeNum) throws DOMException {
    return node.getTextContent() + "-mapped";
  }
}
```

40.3.3. XML Payload Converter

You can also use an implementation of the `org.springframework.integration.xml.XmlPayloadConverter` to provide more granular transformation. The following example shows how to define one:

```
<int-xml:xpath-transformer input-channel="customConverterInput"
                           output-channel="output" xpath-expression="/test/@type"
                           converter="testXmlPayloadConverter" />
```

The following example shows an `XmlPayloadConverter` implementation that works with the preceding example:

```
class TestXmlPayloadConverter implements XmlPayloadConverter {
  public Source convertToSource(Object object) {
    throw new UnsupportedOperationException();
  }
  //
  public Node convertToNode(Object object) {
    try {
      return DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(
          new InputSource(new StringReader("<test type='custom'/>")));
    }
    catch (Exception e) {
      throw new IllegalStateException(e);
    }
  }
  //
  public Document convertToDocument(Object object) {
    throw new UnsupportedOperationException();
  }
}
```

If you do not provide this reference, the `DefaultXmlPayloadConverter` is used. It should suffice in most cases, because it can convert from `Node`, `Document`, `Source`, `File`, `String`, `InputStream`, and `byte[]` payloads. If you need to extend beyond the capabilities of that default implementation, an upstream `Transformer` is probably a better option than providing a reference to a custom implementation of this strategy here.

40.4. Splitting XML Messages

`XPathMessageSplitter` supports messages with either `String` or `Document` payloads. The splitter uses the provided XPath expression to split the payload into a number of nodes. By default, this results in each `Node` instance becoming the payload of a new message. When each message should be a

`Document`, you can set the `createDocuments` flag. Where a `String` payload is passed in, the payload is converted and then split before being converted back to a number of `String` messages. The XPath splitter implements `MessageHandler` and should therefore be configured in conjunction with an appropriate endpoint (see the namespace support example after the following example for a simpler configuration alternative). The following example configures a bean that uses an `XPathMessageSplitter`:

```
<bean id="splittingEndpoint"
      class="org.springframework.integration.endpoint.EventDrivenConsumer">
    <constructor-arg ref="orderChannel" />
    <constructor-arg>
        <bean class="
org.springframework.integration.xml.splitter.XPathMessageSplitter">
            <constructor-arg value="/order/items" />
            <property name="documentBuilder" ref="customisedDocumentBuilder" />
            <property name="outputChannel" ref="orderItemsChannel" />
        </bean>
    </constructor-arg>
</bean>
```

XPath splitter namespace support lets you create a message endpoint with an input channel and output channel, as the following example shows:

```
<!-- Split the order into items and create a new message for each item node -->
<int-xml:xpath-splitter id="orderItemSplitter"
                        input-channel="orderChannel"
                        output-channel="orderItemsChannel">
    <int-xml:xpath-expression expression="/order/items"/>
</int-xml:xpath-splitter>

<!-- Split the order into items, create a new document for each item-->
<int-xml:xpath-splitter id="orderItemDocumentSplitter"
                        input-channel="orderChannel"
                        output-channel="orderItemsChannel"
                        create-documents="true">
    <int-xml:xpath-expression expression="/order/items"/>
    <int:poller fixed-rate="2000"/>
</int-xml:xpath-splitter>
```

Starting with version 4.2, the `XPathMessageSplitter` exposes the `outputProperties` (such as `OutputKeys.OMIT_XML_DECLARATION`) property for an `javax.xml.transform.Transformer` instance when a request `payload` is not of type `org.w3c.dom.Node`. The following example defines a property and uses it with the `output-properties` property:

```
<util:properties id="outputProperties">
    <beans:prop key="#{T (javax.xml.transform.OutputKeys).OMIT_XML_DECLARATION}"
>yes</beans:prop>
</util:properties>

<xpath-splitter input-channel="input"
             output-properties="outputProperties">
    <xpath-expression expression="/orders/order"/>
</xpath-splitter>
```

Starting with `version 4.2`, the `XPathMessageSplitter` exposes an `iterator` option as a `boolean` flag (defaults to `true`). This allows the "streaming" of split nodes in the downstream flow. With the `iterator` mode set to `true`, each node is transformed while iterating. When `false`, all entries are first transformed, before the split nodes start being sent to the output channel. (You can think of the difference as "transform, send, transform, send" versus "transform, transform, send, send".) See Splitter for more information.

40.5. Routing XML Messages with XPath

Similar to SpEL-based routers, Spring Integration provides support for routing messages based on XPath expressions, which lets you create a message endpoint with an input channel but no output channel. Instead, one or more output channels are determined dynamically. The following example shows how to create such a router:

```
<int-xml:xpath-router id="orderTypeRouter" input-channel="orderChannel">
    <int-xml:xpath-expression expression="/order/type"/>
</int-xml:xpath-router>
```

For an overview of attributes that are common among Routers, see Common Router Parameters.

Internally, XPath expressions are evaluated as type `NODESET` and converted to a `List<String>` that represents channel names. Typically, such a list contains a single channel name. However, based on the results of an XPath Expression, the XPath router can also take on the characteristics of a recipient list router if the XPath expression returns more than one value. In that case, the `List<String>` contains more than one channel name. Consequently, messages are sent to all the channels in the list.

Thus, assuming that the XML file passed to the following router configuration contains many `responder` sub-elements that represent channel names, the message is sent to all of those channels:

```
<!-- route the order to all responders-->
<int-xml:xpath-router id="responderRouter" input-channel="orderChannel">
    <int-xml:xpath-expression expression="/request/responders"/>
</int-xml:xpath-router>
```

If the returned values do not represent the channel names directly, you can specify additional mapping parameters to map those returned values to actual channel names. For example if the `/request/responders` expression results in two values (`responderA` and `responderB`), but you do not want to couple the responder names to channel names, you can provide additional mapping configuration, such as the following:

```
<!-- route the order to all responders-->
<int-xml:xpath-router id="responderRouter" input-channel="orderChannel">
    <int-xml:xpath-expression expression="/request/responders"/>
    <int-xml:mapping value="responderA" channel="channelA"/>
    <int-xml:mapping value="responderB" channel="channelB"/>
</int-xml:xpath-router>
```

As already mentioned, the default evaluation type for XPath expressions is `NODESET`, which is converted to a `List<String>` of channel names, which handles single channel scenarios as well as multiple channel scenarios.

Nonetheless, certain XPath expressions may evaluate as type `String` from the very beginning. Consider, for example, the following XPath Expression:

```
name(./node())
```

This expression returns the name of the root node. If the default evaluation type `NODESET` is being used, it results in an exception.

For these scenarios, you can use the `evaluate-as-string` attribute, which lets you manage the evaluation type. It is `FALSE` by default. However, if you set it to `TRUE`, the `String` evaluation type is used.

XPath 1.0 specifies 4 data types:

- Node-sets
- Strings
- Number
- Boolean

When the XPath Router evaluates expressions by using the optional `evaluate-as-string` attribute, the return value is determined by the `string()` function, as defined in the XPath specification. This means that, if the expression selects multiple nodes, it return the string value of the first node.

For further information, see:

- Specification: XML Path Language (XPath) Version 1.0
- XPath specification - string() function

For example, if we want to route based on the name of the root node, we can use the following configuration:

```
<int-xml:xpath-router id="xpathRouterAsString"
        input-channel="xpathStringChannel"
        evaluate-as-string="true">
    <int-xml:xpath-expression expression="name(./node())"/>
</int-xml:xpath-router>
```

40.5.1. XML Payload Converter

For XPath Routers, you can also specify the Converter to use when converting payloads prior to XPath evaluation. As such, the XPath Router supports custom implementations of the `XmlPayloadConverter` strategy, and when configuring an `xpath-router` element in XML, a reference to such an implementation may be provided via the `converter` attribute.

If this reference is not explicitly provided, the `DefaultXmlPayloadConverter` is used. It should be sufficient in most cases, since it can convert from Node, Document, Source, File, and String typed payloads. If you need to extend beyond the capabilities of that default implementation, then an upstream Transformer is generally a better option in most cases, rather than providing a reference to a custom implementation of this strategy here.

40.6. XPath Header Enricher

The XPath header enricher defines a header enricher message transformer that evaluates an XPath expression against the message payload and inserts the result of the evaluation into a message header.

The following listing shows all the available configuration parameters:

```
<int-xml:xpath-header-enricher default-overwrite="true"    ①
                               id=""                       ②
                               input-channel=""            ③
                               output-channel=""           ④
                               should-skip-nulls="true">   ⑤
    <int:poller></int:poller>                              ⑥
    <int-xml:header name=""                                ⑦
                    evaluation-type="STRING_RESULT"        ⑧
                    header-type="int"                      ⑨
                    overwrite="true"                       ⑩
                    xpath-expression=""                    ⑪
                    xpath-expression-ref=""/>              ⑫
</int-xml:xpath-header-enricher>
```

① Specifies the default boolean value for whether to overwrite existing header values. This takes effect only for child elements that do not provide their own 'overwrite' attribute. If you do not set the 'default- overwrite' attribute, the specified header values do not overwrite any existing ones with the same header names. Optional.

② ID for the underlying bean definition. Optional.

③ The receiving message channel of this endpoint. Optional.

④ Channel to which enriched messages are sent. Optional.

⑤ Specifies whether null values, such as might be returned from an expression evaluation, should be skipped. The default value is `true`. If a null value should trigger removal of the corresponding header, set this to `false`. Optional.

⑥ A poller to use with the header enricher. Optional.

⑦ The name of the header to be enriched. Mandatory.

⑧ The result type expected from the XPath evaluation. If you did not set a `header-type` attribute, this is the type of the header value. The following values are allowed: `BOOLEAN_RESULT`, `STRING_RESULT`, `NUMBER_RESULT`, `NODE_RESULT`, and `NODE_LIST_RESULT`. If not set, it defaults internally to `XPathEvaluationType.STRING_RESULT`. Optional.

⑨ The fully qualified class name for the header value type. The result of the XPath evaluation is converted to this type by `ConversionService`. This allows, for example, a `NUMBER_RESULT` (a double) to be converted to an `Integer`. The type can be declared as a primitive (such as `int`), but the result is always the equivalent wrapper class (such as `Integer`). The same integration `ConversionService` discussed in Payload Type Conversion is used for the conversion, so conversion to custom types is supported by adding a custom converter to the service. Optional.

⑩ Boolean value to indicate whether this header value should overwrite an existing header value for the same name if already present on the input `Message`.

⑪ The XPath expression as a `String`. You must set either this attribute or `xpath-expression-ref`, but not both.

⑫ The XPath expression reference. You must set either this attribute or `xpath-expression`, but not both.

40.7. Using the XPath Filter

This component defines an XPath-based message filter. Internally, this components uses a `MessageFilter` that wraps an instance of `AbstractXPathMessageSelector`.

See Filter for further details.

to use the XPath filter you must, at a minimum, provide an XPath expression either by declaring the `xpath-expression` element or by referencing an XPath Expression in the `xpath-expression-ref` attribute.

If the provided XPath expression evaluates to a `boolean` value, no further configuration parameters are necessary. However, if the XPath expression evaluates to a `String`, you should set the `match-value` attribute, against which the evaluation result is matched.

`match-type` has three options:

- `exact`: Correspond to `equals` on `java.lang.String`. The underlying implementation uses a `StringValueTestXPathMessageSelector`
- `case-insensitive`: Correspond to `equals-ignore-case` on `java.lang.String`. The underlying implementation uses a `StringValueTestXPathMessageSelector`
- `regex`: Matches operations one `java.lang.String`. The underlying implementation uses a `RegexTestXPathMessageSelector`

When providing a 'match-type' value of 'regex', the value provided with the `match-value` attribute must be a valid regular expression.

The following example shows all the available attributes for the `xpath-filter` element:

```
<int-xml:xpath-filter discard-channel=""                          ①
                      id=""                                       ②
                      input-channel=""                            ③
                      match-type="exact"                          ④
                      match-value=""                              ⑤
                      output-channel=""                           ⑥
                      throw-exception-on-rejection="false"        ⑦
                      xpath-expression-ref="">                    ⑧
    <int-xml:xpath-expression ... />                              ⑨
    <int:poller ... />                                            ⑩
</int-xml:xpath-filter>
```

① Message channel where you want rejected messages to be sent. Optional.

② ID for the underlying bean definition. Optional.

③ The receiving message channel of this endpoint. Optional.

④ Type of match to apply between the XPath evaluation result and the `match-value`. The default is `exact`. Optional.

⑤ String value to be matched against the XPath evaluation result. If you do not set this attribute, the XPath evaluation must produce a boolean result. Optional.

⑥ The channel to which messages that matched the filter criteria are dispatched. Optional.

⑦ By default, this property is set to `false` and rejected messages (messages that did not match the filter criteria) are silently dropped. However, if set to `true`, message rejection results in an error condition and an exception being propagated upstream to the caller. Optional.

⑧ Reference to an XPath expression instance to evaluate.

⑨ This child element sets the XPath expression to be evaluated. If you do not include this element, you must set the `xpath-expression-ref` attribute. Also, you can include only one `xpath-expression` element.

⑩ A poller to use with the XPath filter. Optional.

40.8. #xpath SpEL Function

Spring Integration, since version 3.0, provides the built-in `#xpath` SpEL function, which invokes the `XPathUtils.evaluate(…)` static method. This method delegates to an `org.springframework.xml.xpath.XPathExpression`. The following listing shows some usage examples:

```
<transformer expression="#xpath(payload, '/name')"/>

<filter expression="#xpath(payload, headers.xpath, 'boolean')"/>

<splitter expression="#xpath(payload, '//book', 'document_list')"/>

<router expression="#xpath(payload, '/person/@age', 'number')">
    <mapping channel="output1" value="16"/>
    <mapping channel="output2" value="45"/>
</router>
```

The `#xpath()` also supports a third optional parameter for converting the result of the XPath evaluation. It can be one of the String constants (`string`, `boolean`, `number`, `node`, `node_list` and `document_list`) or an `org.springframework.xml.xpath.NodeMapper` instance. By default, the `#xpath` SpEL function returns a `String` representation of the XPath evaluation.

To enable the `#xpath` SpEL function, you can add the `spring-integration-xml.jar` to the classpath. You need no declare any components from the Spring Integration XML Namespace.

For more information, see "`Spring Expression Language (SpEL).

40.9. XML Validating Filter

The XML Validating Filter lets you validate incoming messages against provided schema instances. The following schema types are supported:

- xml-schema (www.w3.org/2001/XMLSchema)
- relax-ng (relaxng.org)

Messages that fail validation can either be silently dropped or be forwarded to a definable `discard-channel`. Furthermore, you can configure this filter to throw an `Exception` in case validation fails.

The following listing shows all the available configuration parameters:

```
<int-xml:validating-filter discard-channel=""                    ①
                           id=""                                 ②
                           input-channel=""                      ③
                           output-channel=""                     ④
                           schema-location=""                    ⑤
                           schema-type="xml-schema"              ⑥
                           throw-exception-on-rejection="false"  ⑦
                           xml-converter=""                      ⑧
                           xml-validator="">                     ⑨
    <int:poller .../>                                            ⑩
</int-xml:validating-filter>
```

① Message channel where you want rejected messages to be sent. Optional.

② ID for the underlying bean definition. Optional.

③ The receiving message channel of this endpoint. Optional.

④ Message channel where you want accepted messages to be sent. Optional.

⑤ Sets the location of the schema to validate the message's payload against. Internally uses the `org.springframework.core.io.Resource` interface. You can set this attribute or the `xml-validator` attribute but not both. Optional.

⑥ Sets the schema type. Can be either `xml-schema` or `relax-ng`. Optional. If not set, it defaults to `xml-schema`, which internally translates to `org.springframework.xml.validation.XmlValidatorFactory#SCHEMA_W3C_XML`.

⑦ If `true`, a `MessageRejectedException` is thrown if validation fails for the provided Message's payload. Defaults to `false` if not set. Optional.

⑧ Reference to a custom `org.springframework.integration.xml.XmlPayloadConverter` strategy. Optional.

⑨ Reference to a custom `sorg.springframework.xml.validation.XmlValidator` strategy. You can set this attribute or the `schema-location` attribute but not both. Optional.

⑩ A poller to use with the XPath filter. Optional.

Chapter 41. XMPP Support

Spring Integration provides channel adapters for XMPP. XMPP stands for “Extensible Messaging and Presence Protocol”.

XMPP describes a way for multiple agents to communicate with each other in a distributed system. The canonical use case is to send and receive chat messages, though XMPP can be (and is) used for other kinds of applications. XMPP describes a network of actors. Within that network, actors may address each other directly and broadcast status changes (such as “presence”).

XMPP provides the messaging fabric that underlies some of the biggest instant messaging networks in the world, including Google Talk (GTalk, which is also available from within GMail) and Facebook Chat. Many good open-source XMPP servers are available. Two popular implementations are Openfire and ejabberd.

Spring integration provides support for XMPP by providing XMPP adapters, which support sending and receiving both XMPP chat messages and presence changes from other entries in a client’s roster.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-xmpp</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-xmpp:5.3.0.RC1"
```

As with other adapters, the XMPP adapters come with support for a convenient namespace-based configuration. To configure the XMPP namespace, include the following elements in the headers of your XML configuration file:

```
xmlns:int-xmpp="http://www.springframework.org/schema/integration/xmpp"
xsi:schemaLocation="http://www.springframework.org/schema/integration/xmpp
    https://www.springframework.org/schema/integration/xmpp/spring-integration-
xmpp.xsd"
```

41.1. XMPP Connection

Before using inbound or outbound XMPP adapters to participate in the XMPP network, an actor must establish its XMPP connection. All XMPP adapters connected to a particular account can share this connection object. Typically this requires (at a minimum) `user`, `password`, and `host`. To create a basic XMPP connection, you can use the convenience of the namespace, as the following example shows:

```
<int-xmpp:xmpp-connection
    id="myConnection"
    user="user"
    password="password"
    host="host"
    port="port"
    resource="theNameOfTheResource"
    subscription-mode="accept_all"/>
```

For added convenience, you can rely on the default naming convention and omit the `id` attribute. The default name (`xmppConnection`) is used for this connection bean.

If the XMPP connection goes stale, reconnection attempts are made with an automatic login as long as the previous connection state was logged (authenticated). We also register a `ConnectionListener`, which logs connection events if the `DEBUG` logging level is enabled.

The `subscription-mode` attribute initiates the roster listener to deal with incoming subscriptions from other users. This functionality is not always available for the target XMPP servers. For example, Google Cloud Messaging (GCM) and Firebase Cloud Messaging (FCM) fully disable it. To switch off the roster listener for subscriptions, you can configure it with an empty string when using XML configuration (`subscription-mode=""`) or with `XmppConnectionFactoryBean.setSubscriptionMode(null)` when using Java Configuration. Doing so disables the roster at the login phase as well. See `Roster.setRosterLoadedAtLogin(boolean)` for more information.

41.2. XMPP Messages

Spring Integration provides support for sending and receiving XMPP messages. For receiving them, it offers an inbound message channel adapter. For sending them, it offers an outbound message channel adapter.

41.2.1. Inbound Message Channel Adapter

The Spring Integration adapters support receiving chat messages from other users in the system. To do so, the inbound message channel adapter “logs in” as a user on your behalf and receives the messages sent to that user. Those messages are then forwarded to your Spring Integration client.

The `inbound-channel-adapter` element provides Configuration support for the XMPP inbound message channel adapter. The following example shows how to configure it:

```
<int-xmpp:inbound-channel-adapter id="xmppInboundAdapter"
    channel="xmppInbound"
    xmpp-connection="testConnection"
    payload-expression="getExtension('google:mobile:data').json"
    stanza-filter="stanzaFilter"
    auto-startup="true"/>
```

Along with the usual attributes (for a message channel adapter), this adapter also requires a reference to an XMPP Connection.

The XMPP inbound adapter is event-driven and a `Lifecycle` implementation. When started, it registers a `PacketListener` that listens for incoming XMPP chat messages. It forwards any received messages to the underlying adapter, which converts them to Spring Integration messages and sends them to the specified `channel`. When stopped, it unregisters the `PacketListener`.

Starting with version 4.3, the `ChatMessageListeningEndpoint` (and its `<int-xmpp:inbound-channel-adapter>`) supports the injection of a `org.jivesoftware.smack.filter.StanzaFilter` to be registered on the provided `XMPPConnection`, together with an internal `StanzaListener` implementation. See the Javadoc for more information.

Version 4.3 introduced the `payload-expression` attribute for the `ChatMessageListeningEndpoint`. The incoming `org.jivesoftware.smack.packet.Message` represents a root object for the evaluation context. This option is useful when you use XMPP extensions. For example, for the GCM protocol we can extract the body by using the following expression:

```
payload-expression="getExtension('google:mobile:data').json"
```

The following example extracts the body for the XHTML protocol:

```
payload-
expression="getExtension(T(org.jivesoftware.smackx.xhtmlim.packet.XHTMLExtension).
NAMESPACE).bodies[0]"
```

To simplify access to the extension in the XMPP Message, the `extension` variable is added into the `EvaluationContext`. Note that it is added when only one extension is present in the message. The preceding examples that show the `namespace` manipulations can be simplified to the following example:

```
payload-expression="#extension.json"
payload-expression="#extension.bodies[0]"
```

41.2.2. Outbound Message Channel Adapter

You can also send chat messages to other users on XMPP by using the outbound message channel adapter. The `outbound-channel-adapter` element provides configuration support for the XMPP outbound message channel adapter.

```
<int-xmpp:outbound-channel-adapter id="outboundEventAdapter"
                          channel="outboundEventChannel"
                          xmpp-connection="testConnection"/>
```

The adapter expects its input to be (at a minimum) a payload of type `java.lang.String` and a header value for `XmppHeaders.CHAT_TO` that specifies to which user the message should be sent. To create a message, you can use Java code similar to the following:

```
Message<String> xmppOutboundMsg = MessageBuilder.withPayload("Hello, XMPP!" )
                          .setHeader(XmppHeaders.CHAT_TO, "userhandle")
                          .build();
```

You can also set the header by using the XMPP header-enricher support, as the following example shows:

```
<int-xmpp:header-enricher input-channel="input" output-channel="output">
    <int-xmpp:chat-to value="test1@example.org"/>
</int-xmpp:header-enricher>
```

Starting with version 4.3, the packet extension support has been added to the `ChatMessageSendingMessageHandler` (the `<int-xmpp:outbound-channel-adapter>` in XML configuration). Along with the regular `String` and `org.jivesoftware.smack.packet.Message` payload, now you can send a message with a payload of `org.jivesoftware.smack.packet.ExtensionElement` (which is populated to the `org.jivesoftware.smack.packet.Message.addExtension()`) instead of `setBody()`. For convenience, we added an `extension-provider` option for the `ChatMessageSendingMessageHandler`. It lets you inject `org.jivesoftware.smack.provider.ExtensionElementProvider`, which builds an `ExtensionElement` against the payload at runtime. For this case, the payload must be a string in JSON or XML format, depending of the XEP protocol.

41.3. XMPP Presence

XMPP also supports broadcasting state. You can use this ability to let people who have you on their roster see your state changes. This happens all the time with your IM clients. You change your away status and set an away message, and everybody who has you on their roster sees your icon or username change to reflect this new state and might see your new “away” message. If you would like to receive notifications or notify others of state changes, you can use Spring Integration’s “presence” adapters.

41.3.1. Inbound Presence Message Channel Adapter

Spring Integration provides an inbound presence message channel adapter, which supports receiving presence events from other users in the system who are on your roster. To do this, the adapter “logs in” as a user on your behalf, registers a `RosterListener`, and forwards received presence update events as messages to the channel identified by the `channel` attribute. The payload of the message is a `org.jivesoftware.smack.packet.Presence` object (see www.igniterealtime.org/builds/smack/docs/latest/javadoc/org/jivesoftware/smack/packet/Presence.html).

The `presence-inbound-channel-adapter` element provides configuration support for the XMPP inbound presence message channel adapter. The following example configures an inbound presence message channel adapter:

```
<int-xmpp:presence-inbound-channel-adapter channel="outChannel"
        xmpp-connection="testConnection" auto-startup="false"/>
```

Along with the usual attributes, this adapter requires a reference to an XMPP Connection. This adapter is event-driven and a `Lifecycle` implementation. It registers a `RosterListener` when started and unregisters that `RosterListener` when stopped.

41.3.2. Outbound Presence Message Channel Adapter

Spring Integration also supports sending presence events to be seen by other users in the network who happen to have you on their roster. When you send a message to the outbound presence message channel adapter, it extracts the payload (which is expected to be of type `org.jivesoftware.smack.packet.Presence`) and sends it to the XMPP Connection, thus advertising your presence events to the rest of the network.

The `presence-outbound-channel-adapter` element provides configuration support for the XMPP outbound presence message channel adapter. The following example shows how to configure an outbound presence message channel adapter:

```
<int-xmpp:presence-outbound-channel-adapter id="eventOutboundPresenceChannel"
    xmpp-connection="testConnection"/>
```

It can also be a polling consumer (if it receives messages from a pollable channel) in which case you would need to register a poller. The following example shows how to do so:

```
<int-xmpp:presence-outbound-channel-adapter id="pollingOutboundPresenceAdapter"
        xmpp-connection="testConnection"
        channel="pollingChannel">
    <int:poller fixed-rate="1000" max-messages-per-poll="1"/>
</int-xmpp:presence-outbound-channel-adapter>
```

Like its inbound counterpart, it requires a reference to an XMPP Connection.

If you rely on the default naming convention for an XMPP Connection bean (described earlier) and you have only one XMPP Connection bean configured in your application context, you can omit the `xmpp-connection` attribute. In that case, the bean with named `xmppConnection` is located and injected into the adapter.

41.4. Advanced Configuration

Spring Integration's XMPP support is based on the Smack 4.0 API (www.igniterealtime.org/projects/smack/), which allows more complex configuration of the XMPP Connection object.

As stated earlier, the `xmpp-connection` namespace support is designed to simplify basic connection configuration and supports only a few common configuration attributes. However, the `org.jivesoftware.smack.ConnectionConfiguration` object defines about 20 attributes, and adding namespace support for all of them offers no real value. So, for more complex connection configurations, you can configure an instance of our `XmppConnectionFactoryBean` as a regular bean and inject a `org.jivesoftware.smack.ConnectionConfiguration` as a constructor argument to that `FactoryBean`. You can specify every property you need directly on that `ConnectionConfiguration` instance. (A bean definition with the 'p' namespace would work well.) This way, you can directly set SSL (or any other attributes). The following example shows how to do so:

```
<bean id="xmppConnection" class="o.s.i.xmpp.XmppConnectionFactoryBean">
    <constructor-arg>
        <bean class="org.jivesoftware.smack.ConnectionConfiguration">
            <constructor-arg value="myServiceName"/>
            <property name="socketFactory" ref="..."/>
        </bean>
    </constructor-arg>
</bean>

<int:channel id="outboundEventChannel"/>

<int-xmpp:outbound-channel-adapter id="outboundEventAdapter"
    channel="outboundEventChannel"
    xmpp-connection="xmppConnection"/>
```

The Smack API also offers static initializers, which can be helpful. For more complex cases (such as registering a SASL mechanism), you may need to execute certain static initializers. One of those static initializers is `SASLAuthentication`, which lets you register supported SASL mechanisms. For that level of complexity, we recommend using Spring Java configuration for the XMPP connection configuration. That way, you can configure the entire component through Java code and execute all other necessary Java code, including static initializers, at the appropriate time. The following exampe shows how to configure an XMPP connection with an SASL (Simple Authentication and Security Layer) in Java:

```
@Configuration
public class CustomConnectionConfiguration {
  @Bean
  public XMPPConnection xmppConnection() {
    SASLAuthentication.supportSASLMechanism("EXTERNAL", 0); // static initializer

    ConnectionConfiguration config = new ConnectionConfiguration("localhost",
5223);
    config.setTrustorePath("path_to_truststore.jks");
    config.setSecurityEnabled(true);
    config.setSocketFactory(SSLSocketFactory.getDefault());
    return new XMPPConnection(config);
  }
}
```

For more information on using Java for application context configuration, see the following section in the Spring Reference Manual.

41.5. XMPP Message Headers

The Spring Integration XMPP Adapters automatically map standard XMPP properties. By default, these properties are copied to and from Spring Integration `MessageHeaders` by using `DefaultXmppHeaderMapper`.

Any user-defined headers are not copied to or from an XMPP Message, unless explicitly specified by the `requestHeaderNames` or `replyHeaderNames` properties of the `DefaultXmppHeaderMapper`.

When mapping user-defined headers, the values can also contain simple wildcard patterns (such "thing*" or "*thing").

Starting with version 4.1, `AbstractHeaderMapper` (a superclass of `DefaultXmppHeaderMapper`) lets you configure the `NON_STANDARD_HEADERS` token for the `requestHeaderNames` property (in addition to `STANDARD_REQUEST_HEADERS`), to map all user-defined headers.

The `org.springframework.xmpp.XmppHeaders` class identifies the default headers to be used by the `DefaultXmppHeaderMapper`:

- `xmpp_from`
- `xmpp_subject`
- `xmpp_thread`
- `xmpp_to`
- `xmpp_type`

Starting with version 4.3, you can negate patterns in the header mappings by preceding the pattern with `!`. Negated patterns get priority, so a list such as `STANDARD_REQUEST_HEADERS,thing1,thing*,!thing2,!thing3,qux,!thing1` does not map `thing1`, `thing2`,or `thing3`. That list does map the standard headers plus `thing4` and `qux`.

If you have a user-defined header that begins with `!` that you do wish to map, can escape it with `\` thus: `STANDARD_REQUEST_HEADERS,\!myBangHeader`. In that example, the standard request headers and `!myBangHeader` are mapped.

41.6. XMPP Extensions

Extensions put the “Extensible” in the “Extensible Messaging and Presence Protocol”.

XMPP is based around XML, a data format that supports a concept known as namespacing. Through namespacing, you can add bits to XMPP that are not defined in the original specifications. The XMPP specification deliberately describes only a set of core features:

- How a client connects to a server
- Encryption (SSL/TLS)
- Authentication
- How servers can communicate with each other to relay messages

- A few other basic building blocks

Once you have implemented this, you have an XMPP client and can send any kind of data you like. However, you may need to do more than the basics. For example, you might need to include formatting (bold, italic, and so on) in a message, which is not defined in the core XMPP specification. Well, you can make up a way to do that, but, unless everyone else does it the same way you do, no other software can interpret it (they ignore namespaces they cannot understand).

To solve that problem, the XMPP Standards Foundation (XSF) publishes a series of extra documents, known as XMPP Enhancement Proposals (XEPs). In general, each XEP describes a particular activity (from message formatting to file transfers, multi-user chats, and many more). They also provide a standard format for everyone to use for that activity.

The Smack API provides many XEP implementations with its `extensions` and `experimental` projects. Starting with Spring Integration version 4.3, you can use any XEP with the existing XMPP channel adapters.

To be able to process XEPs or any other custom XMPP extensions, you must provide the Smack's `ProviderManager` pre-configuration. You can do so with `static` Java code, as the following example shows:

```
ProviderManager.addIQProvider("element", "namespace", new MyIQProvider());
ProviderManager.addExtensionProvider("element", "namespace", new MyExtProvider());
```

You can also use a `.providers` configuration file in the specific instance and access it with a JVM argument, as the following example shows:

```
-Dsmack.provider.file=file:///c:/my/provider/mycustom.providers
```

The `mycustom.providers` file might be as follows:

```
<?xml version="1.0"?>
<smackProviders>
<iqProvider>
    <elementName>query</elementName>
    <namespace>jabber:iq:time</namespace>
    <className>org.jivesoftware.smack.packet.Time</className>
</iqProvider>

<iqProvider>
    <elementName>query</elementName>
    <namespace>https://jabber.org/protocol/disco#items</namespace>
    <className>org.jivesoftware.smackx.provider.DiscoverItemsProvider</className>
</iqProvider>

<extensionProvider>
    <elementName>subscription</elementName>
    <namespace>https://jabber.org/protocol/pubsub</namespace>
    <className>
org.jivesoftware.smackx.pubsub.provider.SubscriptionProvider</className>
</extensionProvider>
</smackProviders>
```

For example, the most popular XMPP messaging extension is Google Cloud Messaging (GCM). The Smack library provides `org.jivesoftware.smackx.gcm.provider.GcmExtensionProvider` for that purposes. By default, it registers that class with the `smack-experimental` jar in the classpath by using the `experimental.providers` resource, as the following Maven example shows:

```
<!-- GCM JSON payload -->
<extensionProvider>
    <elementName>gcm</elementName>
    <namespace>google:mobile:data</namespace>
    <className>
org.jivesoftware.smackx.gcm.provider.GcmExtensionProvider</className>
</extensionProvider>
```

Also, the `GcmPacketExtension` lets the target messaging protocol parse incoming packets and build outgoing packets, as the following examples show:

```
GcmPacketExtension gcmExtension = (GcmPacketExtension) xmppMessage.getExtension
(GcmPacketExtension.NAMESPACE);
String message = gcmExtension.getJson());
```

```
GcmPacketExtension packetExtension = new GcmPacketExtension(gcmJson);
Message smackMessage = new Message();
smackMessage.addExtension(packetExtension);
```

See Inbound Message Channel Adapter and Outbound Message Channel Adapter earlier in this chapter for more information.

Chapter 42. Zookeeper Support

Version 4.2 added Zookeeper support to the framework in version 4.2, which consists of:

- A metadata store
- A lock registry
- Leadership event handling

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-zookeeper</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-zookeeper:5.3.0.RC1"
```

42.1. Zookeeper Metadata Store

You ca use the `ZookeeperMetadataStore` where any `MetadataStore` is needed, such as for persistent file list filters. See Metadata Store for more information. The following example configures a Zookeeper metadata store with XML:

```
<bean id="client" class=
"org.springframework.integration.zookeeper.config.CuratorFrameworkFactoryBean">
    <constructor-arg value="${connect.string}" />
</bean>

<bean id="meta" class=
"org.springframework.integration.zookeeper.metadata.ZookeeperMetadataStore">
    <constructor-arg ref="client" />
</bean>
```

The following example shows how to configure a Zookeeper metadata store with Java:

```
@Bean
public MetadataStore zkStore(CuratorFramework client) {
    return new ZookeeperMetadataStore(client);
}
```

42.2. Zookeeper Lock Registry

The `ZookeeperLockRegistry` can be used where any `LockRegistry` is needed, such as when using an aggregator in a clustered environment with a shared `MessageStore`.

A `LockRegistry` is used to "look up" a lock based on a key (the aggregator uses `correlationId`). By default, locks in the `ZookeeperLockRegistry` are maintained in zookeeper under the following path: `/SpringIntegration-LockRegistry/`. You can customize the path by providing an implementation of `ZookeeperLockRegistry.KeyToPathStrategy`, as the following example shows:

```
public interface KeyToPathStrategy {

    String pathFor(String key);

    boolean bounded();

}
```

If the strategy returns `true` from `isBounded`, unused locks do not need to be harvested. For unbounded strategies (such as the default), you need to periodically invoke `expireUnusedOlderThan(long age)` to remove old unused locks from memory.

42.3. Zookeeper Leadership Event Handling

The following example uses XML to configure an application for leader election in Zookeeper:

```
<int-zk:leader-listener client="client" path="/siNamespace" role="cluster" />
```

`client` is a reference to a `CuratorFramework` bean. A `CuratorFrameworkFactoryBean` is available. When a leader is elected, an `OnGrantedEvent` is published for the role `cluster`. Any endpoints in that role are started. When leadership is revoked, an `OnRevokedEvent` is published for the role `cluster`. Any endpoints in that role are stopped. See Endpoint Roles for more information.

You can use Java configuration to create an instance of the leader initiator, as the following example shows:

```
@Bean
public LeaderInitiatorFactoryBean leaderInitiator(CuratorFramework client) {
    return new LeaderInitiatorFactoryBean()
                .setClient(client)
                .setPath("/siTest/")
                .setRole("cluster");
}
```

Starting with version 5.3, a `candidate` option is exposed on the `LeaderInitiatorFactoryBean` for more configuration control of the externally provided `Candidate` instance. Only one of the `candidate` or `role` options has to be provided, but not both; the `role` options creates internally a `DefaultCandidate` instance with an `UUID` for `id` option.

Appendices

The appendices cover advanced topics and additional resources.

The last appendix covers the history of Spring Integration.

Appendix A: Error Handling

As described in the overview at the very beginning of this manual, one of the main motivations behind a message-oriented framework such as Spring Integration is to promote loose coupling between components. The message channel plays an important role, in that producers and consumers do not have to know about each other. However, the advantages also have some drawbacks. Some things become more complicated in a loosely coupled environment, and one example is error handling.

When sending a message to a channel, the component that ultimately handles that message may or may not be operating within the same thread as the sender. If using a simple default `DirectChannel` (when the `<channel>` element that has no `<queue>` child element and no 'task-executor' attribute), the message handling occurs in the same thread that sends the initial message. In that case, if an `Exception` is thrown, it can be caught by the sender (or it may propagate past the sender if it is an uncaught `RuntimeException`). This is the same behavior as an exception-throwing operation in a normal Java call stack.

A message flow that runs on a caller thread might be invoked through a messaging gateway (see Messaging Gateways) or a `MessagingTemplate` (see `MessagingTemplate`). In either case, the default behavior is to throw any exceptions to the caller. For the messaging gateway, see Error Handling for details about how the exception is thrown and how to configure the gateway to route the errors to an error channel instead. When using a `MessagingTemplate` or sending to a `MessageChannel` directly, exceptions are always thrown to the caller.

When adding asynchronous processing, things become rather more complicated. For instance, if the 'channel' element does provide a 'queue' child element (`QueueChannel` in Java & Annotations Configuration), the component that handles the message operates in a different thread than the sender. The same is true when an `ExecutorChannel` is used. The sender may have dropped the `Message` into the channel and moved on to other things. There is no way for the `Exception` to be thrown directly back to that sender by using standard `Exception` throwing techniques. Instead, handling errors for asynchronous processes requires that the error-handling mechanism also be asynchronous.

Spring Integration supports error handling for its components by publishing errors to a message channel. Specifically, the `Exception` becomes the payload of a Spring Integration `ErrorMessage`. That `Message` is then sent to a message channel that is resolved in a way that is similar to the 'replyChannel' resolution. First, if the request `Message` being handled at the time the `Exception` occurred contains an 'errorChannel' header (the header name is defined in the `MessageHeaders.ERROR_CHANNEL` constant), the `ErrorMessage` is sent to that channel. Otherwise, the error handler sends to a “global” channel whose bean name is `errorChannel` (this is also defined as a constant: `IntegrationContextUtils.ERROR_CHANNEL_BEAN_NAME`).

A default `errorChannel` bean is created internally by the Framework. However, you can define your own if you want to control the settings. The following example shows how to define an error channel in XML configuration backed by a queue with a capacity of `500`:

```
<int:channel id="errorChannel">
    <int:queue capacity="500"/>
</int:channel>
```

The default error channel is a `PublishSubscribeChannel`.

The most important thing to understand here is that the messaging-based error handling applies only to exceptions that are thrown by a Spring Integration task that is executing within a `TaskExecutor`. This does not apply to exceptions thrown by a handler that operates within the same thread as the sender (for example, through a `DirectChannel` as described earlier in this section).

When exceptions occur in a scheduled poller task's execution, those exceptions are wrapped in `ErrorMessage` instances and sent to the 'errorChannel' as well.

To enable global error handling, register a handler on that channel. For example, you can configure Spring Integration's `ErrorMessageExceptionTypeRouter` as the handler of an endpoint that is subscribed to the 'errorChannel'. That router can then spread the error messages across multiple channels, based on the `Exception` type.

Starting with version 4.3.10, Spring Integration provides the `ErrorMessagePublisher` and the `ErrorMessageStrategy`. You can use them as a general mechanism for publishing `ErrorMessage` instances. You can call or extend them in any error handling scenarios. The `ErrorMessageSendingRecoverer` extends this class as a `RecoveryCallback` implementation that can be used with retry, such as the `RequestHandlerRetryAdvice`. The `ErrorMessageStrategy` is used to build an `ErrorMessage` based on the provided exception and an `AttributeAccessor` context. It can be injected into any `MessageProducerSupport` or `MessagingGatewaySupport`. The `requestMessage` is stored under `ErrorMessageUtils.INPUT_MESSAGE_CONTEXT_KEY` in the `AttributeAccessor` context. The `ErrorMessageStrategy` can use that `requestMessage` as the `originalMessage` property of the `ErrorMessage` it creates. The `DefaultErrorMessageStrategy` does exactly that.

Starting with version 5.2, all the `MessageHandlingException` instances thrown by the framework components, includes a component `BeanDefinition` resource and source to determine a configuration point form the exception. In case of XML configuration, a resource is an XML file path and source an XML tag with its `id` attribute. With Java & Annotation configuration, a resource is a `@Configuration` class and source is a `@Bean` method. In most case the target integration flow solution is based on the out-of-the-box components and their configuration options. When an exception happens at runtime, there is no any end-user code involved in stack trace because an execution is against beans, not their configuration. Including a resource and source of the bean definition helps to determine possible configuration mistakes and provides better developer experience.

Appendix B: Spring Expression Language (SpEL)

You can configure many Spring Integration components by using expressions written in the Spring Expression Language.

In most cases, the `#root` object is the `Message`, which has two properties (`headers` and `payload`) that allow such expressions as `payload`, `payload.thing`, `headers['my.header']`, and so on.

In some cases, additional variables are provided. For example, `<int-http:inbound-gateway/>` provides `#requestParams` (parameters from the HTTP request) and `#pathVariables` (values from path placeholders in the URI).

For all SpEL expressions, a `BeanResolver` is available to enable references to any bean in the application context (for example, `@myBean.foo(payload)`). In addition, two `PropertyAccessors` are available. A `MapAccessor` enables accessing values in a `Map` by using a key and a `ReflectivePropertyAccessor`, which allows access to fields and JavaBean compliant properties (by using getters and setters). This is how you can access the `Message` headers and payload properties.

B.1. SpEL Evaluation Context Customization

Starting with Spring Integration 3.0, you can add additional `PropertyAccessor` instances to the SpEL evaluation contexts used by the framework. The framework provides the (read-only) `JsonPropertyAccessor`, which you can use to access fields from a `JsonNode` or JSON in a `String`. You can also create your own `PropertyAccessor` if you have specific needs.

In addition, you can add custom functions. Custom functions are `static` methods declared on a class. Functions and property accessors are available in any SpEL expression used throughout the framework.

The following configuration shows how to directly configure the `IntegrationEvaluationContextFactoryBean` with custom property accessors and functions:

```
<bean id="integrationEvaluationContext"
            class=
"org.springframework.integration.config.IntegrationEvaluationContextFactoryBean">
    <property name="propertyAccessors">
        <util:map>
            <entry key="things">
                <bean class="things.MyCustomPropertyAccessor"/>
            </entry>
        </util:map>
    </property>
    <property name="functions">
        <map>
            <entry key="barcalc" value="#{T(things.MyFunctions).getMethod('calc',
T(things.MyThing))}"/>
        </map>
    </property>
</bean>
```

For convenience, Spring Integration provides namespace support for both property accessors and functions, as described in the following sections. The framework automatically configures the factory bean on your behalf.

This factory bean definition overrides the default `integrationEvaluationContext` bean definition. It adds the custom accessor and one custom function to the list (which also includes the standard accessors mentioned earlier).

Note that custom functions are static methods. In the preceding example, the custom function is a static method called `calc` on a class called `MyFunctions` and takes a single parameter of type `MyThing`.

Suppose you have a `Message` with a payload that has a type of `MyThing`. Further suppose that you need to perform some action to create an object called `MyObject` from `MyThing` and then invoke a custom function called `calc` on that object.

The standard property accessors do not know how to get a `MyObject` from a `MyThing`, so you could write and configure a custom property accessor to do so. As a result, your final expression might be `"#barcalc(payload.myObject)"`.

The factory bean has another property (`typeLocator`), which lets you customize the `TypeLocator` used during SpEL evaluation. You might need to do so running in some environments that use a non-standard `ClassLoader`. In the following example, SpEL expressions always use the bean factory's class loader:

```
<bean id="integrationEvaluationContext"
        class=
"org.springframework.integration.config.IntegrationEvaluationContextFactoryBean">
    <property name="typeLocator">
        <bean class=
"org.springframework.expression.spel.support.StandardTypeLocator">
            <constructor-arg value="#{beanFactory.beanClassLoader}"/>
        </bean>
    </property>
</bean>
```

B.2. SpEL Functions

Spring Integration provides namespace support to let you create SpEL custom functions. You can specify `<spel-function/>` components to provide custom SpEL functions to the `EvaluationContext` used throughout the framework. Instead of configuring the factory bean shown earlier, you can add one or more of these components, and the framework automatically adds them to the default `integrationEvaluationContext` factory bean.

For example, suppose you have a useful static method to evaluate XPath. The following example shows how you can create a custom function to use that method:

```
<int:spel-function id="xpath"
    class="com.something.test.XPathUtils" method="evaluate(java.lang.String,
java.lang.Object)"/>

<int:transformer input-channel="in" output-channel="out"
         expression="#xpath('//things/@mythings', payload)" />
```

Given the preceding example:

- The default `IntegrationEvaluationContextFactoryBean` bean with an ID of `integrationEvaluationContext` is registered with the application context.
- The `<spel-function/>` is parsed and added to the `functions Map` of `integrationEvaluationContext` as a map entry with its `id` as the key and the static `Method` as the value.
- The `integrationEvaluationContext` factory bean creates a new `StandardEvaluationContext` instance, and it is configured with the default `PropertyAccessor` instances, a `BeanResolver`, and the custom functions.
- That `EvaluationContext` instance is injected into the `ExpressionEvaluatingTransformer` bean.

To provide a SpEL Function by using Java configuration, you can declare a `SpelFunctionFactoryBean` bean for each function. The following example shows how to create a custom function:

```
@Bean
public SpelFunctionFactoryBean xpath() {
    return new SpelFunctionFactoryBean(XPathUtils.class, "evaluate");
}
```

SpEL functions declared in a parent context are also made available in any child contexts. Each context has its own instance of the `integrationEvaluationContext` factory bean, because each needs a different `BeanResolver`, but the function declarations are inherited and can be overridden by declaring a SpEL function with the same name.

B.2.1. Built-in SpEL Functions

Spring Integration provides the folloiwng standard functions, which are registered with the application context automatically on start up:

- `#jsonPath`: Evaluates a 'jsonPath' on a specified object. This function invokes `JsonPathUtils.evaluate(⋯)`, which delegates to the Jayway JsonPath library. The following listing shows some usage examples:

```
<transformer expression="#jsonPath(payload, '$.store.book[0].author')"/>

<filter expression="#jsonPath(payload,'$..book[2].isbn') matches '\d-\d{3}-
\d{5}-\d'"/>

<splitter expression="#jsonPath(payload, '$.store.book')"/>

<router expression="#jsonPath(payload, headers.jsonPath)">
    <mapping channel="output1" value="reference"/>
    <mapping channel="output2" value="fiction"/>
</router>
```

  `#jsonPath` also supports a third (optional) parameter: an array of `com.jayway.jsonpath.Filter`, which can be provided by a reference to a bean or bean method (for example).

  Using this function requires the Jayway JsonPath library (`json-path.jar`) to be on the classpath. Otherwise the `#jsonPath` SpEL function is not registered.

  For more information regarding JSON see 'JSON Transformers' in Transformer.

- `#xpath`: To evaluate an 'xpath' on some provided object. For more information regarding XML and XPath, see XML Support - Dealing with XML Payloads.

B.3. Property Accessors

Spring Integration provides namespace support to let you create SpEL custom `PropertyAccessor` implementations. You can use the `<spel-property-accessors/>` component to provide a list of custom `PropertyAccessor` instances to the `EvaluationContext` used throughout the framework. Instead of configuring the factory bean shown earlier, you can add one or more of these components, and the framework automatically adds the accessors to the default `integrationEvaluationContext` factory bean. The following example shows how to do so:

```
<int:spel-property-accessors>
    <bean id="jsonPA" class=
"org.springframework.integration.json.JsonPropertyAccessor"/>
    <ref bean="fooPropertyAccessor"/>
</int:spel-property-accessors>
```

In the preceding example, two custom `PropertyAccessor` instances are injected into the `EvaluationContext` (in the order in which they are declared).

To provide `PropertyAccessor` instances by using Java Configuration, you should declare a `SpelPropertyAccessorRegistrar` bean with a name of `spelPropertyAccessorRegistrar` (dictated by the `IntegrationContextUtils.SPEL_PROPERTY_ACCESSOR_REGISTRAR_BEAN_NAME` constant). The following example shows how to configure two custom `PropertyAccessor` instances with Java:

```
@Bean
public SpelPropertyAccessorRegistrar spelPropertyAccessorRegistrar() {
    return new SpelPropertyAccessorRegistrar(new JsonPropertyAccessor())
                    .add(fooPropertyAccessor());
}
```

Custom `PropertyAccessor` instances declared in a parent context are also made available in any child contexts. They are placed at the end of result list (but before the default `org.springframework.context.expression.MapAccessor` and `o.s.expression.spel.support.ReflectivePropertyAccessor`). If you declare a `PropertyAccessor` with the same bean ID in a child context, it overrides the parent accessor. Beans declared within a `<spel-property-accessors/>` must have an 'id' attribute. The final order of usage is as follows:

- The accessors in the current context, in the order in which they are declared
- Any accessors from parent contexts, in order
- The `MapAccessor`
- The `ReflectivePropertyAccessor`

Appendix C: Message Publishing

The (Aspect-oriented Programming) AOP message publishing feature lets you construct and send a message as a by-product of a method invocation. For example, imagine you have a component and, every time the state of this component changes, you want to be notified by a message. The easiest way to send such notifications is to send a message to a dedicated channel, but how would you connect the method invocation that changes the state of the object to a message sending process, and how should the notification message be structured? The AOP message publishing feature handles these responsibilities with a configuration-driven approach.

C.1. Message Publishing Configuration

Spring Integration provides two approaches: XML configuration and annotation-driven (Java) configuration.

C.1.1. Annotation-driven Configuration with the `@Publisher` Annotation

The annotation-driven approach lets you annotate any method with the `@Publisher` annotation to specify a 'channel' attribute. Starting with version 5.1, to switch this functionality on, you must use the `@EnablePublisher` annotation on some `@Configuration` class. See Configuration and `@EnableIntegration` for more information. The message is constructed from the return value of the method invocation and sent to the channel specified by the 'channel' attribute. To further manage message structure, you can also use a combination of both `@Payload` and `@Header` annotations.

Internally, this message publishing feature of Spring Integration uses both Spring AOP by defining `PublisherAnnotationAdvisor` and the Spring Expression Language (SpEL), giving you considerable flexibility and control over the structure of the `Message` it publishes.

The `PublisherAnnotationAdvisor` defines and binds the following variables:

- `#return`: Binds to a return value, letting you reference it or its attributes (for example, `#return.something`, where 'something' is an attribute of the object bound to `#return`)
- `#exception`: Binds to an exception if one is thrown by the method invocation
- `#args`: Binds to method arguments so that you can extract individual arguments by name (for example, `#args.fname`)

Consider the following example:

```
@Publisher
public String defaultPayload(String fname, String lname) {
  return fname + " " + lname;
}
```

In the preceding example, the message is constructed with the following structure:

- The message payload is the return type and value of the method. This is the default.
- A newly constructed message is sent to a default publisher channel that is configured with an annotation post processor (covered later in this section).

The following example is the same as the preceding example, except that it does not use a default publishing channel:

```
@Publisher(channel="testChannel")
public String defaultPayload(String fname, @Header("last") String lname) {
  return fname + " " + lname;
}
```

Instead of using a default publishing channel, we specify the publishing channel by setting the 'channel' attribute of the `@Publisher` annotation. We also add a `@Header` annotation, which results in the message header named 'last' having the same value as the 'lname' method parameter. That header is added to the newly constructed message.

The following example is almost identical to the preceding example:

```
@Publisher(channel="testChannel")
@Payload
public String defaultPayloadButExplicitAnnotation(String fname, @Header String
lname) {
  return fname + " " + lname;
}
```

The only difference is that we use a `@Payload` annotation on the method to explicitly specify that the return value of the method should be used as the payload of the message.

The following example expands on the previous configuration by using the Spring Expression Language in the `@Payload` annotation to further instruct the framework about how the message should be constructed:

```
@Publisher(channel="testChannel")
@Payload("#return + #args.lname")
public String setName(String fname, String lname, @Header("x") int num) {
  return fname + " " + lname;
}
```

In the preceding example, the message is a concatenation of the return value of the method invocation and the 'lname' input argument. The Message header named 'x' has its value determined by the 'num' input argument. That header is added to the newly constructed message.

```
@Publisher(channel="testChannel")
public String argumentAsPayload(@Payload String fname, @Header String lname) {
  return fname + " " + lname;
}
```

In the preceding example, you see another usage of the `@Payload` annotation. Here, we annotate a method argument that becomes the payload of the newly constructed message.

As with most other annotation-driven features in Spring, you need to register a post-processor (`PublisherAnnotationBeanPostProcessor`). The following example shows how to do so:

```
<bean class=
"org.springframework.integration.aop.PublisherAnnotationBeanPostProcessor"/>
```

For a more concise configuration, you can instead use namespace support, as the following example shows:

```
<int:annotation-config>
    <int:enable-publisher default-publisher-channel="defaultChannel"/>
</int:annotation-config>
```

For Java configuration, you must use the `@EnablePublisher` annotation, as the following example shows:

```
@Configuration
@EnableIntegration
@EnablePublisher("defaultChannel")
public class IntegrationConfiguration {
    ...
}
```

Starting with version 5.1.3, the `<int:enable-publisher>` component, as well as the `@EnablePublisher` annotation have the `proxy-target-class` and `order` attributes for tuning the `ProxyFactory` configuration.

Similar to other Spring annotations (`@Component`, `@Scheduled`, and so on), you can also use `@Publisher` as a meta-annotation. This means that you can define your own annotations that are treated in the same way as the `@Publisher` itself. The following example shows how to do so:

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Publisher(channel="auditChannel")
public @interface Audit {
...
}
```

In the preceding example, we define the `@Audit` annotation, which is itself annotated with `@Publisher`. Also note that you can define a `channel` attribute on the meta-annotation to encapsulate where messages are sent inside of this annotation. Now you can annotate any method with the `@Audit` annotation, as the following example shows:

```
@Audit
public String test() {
    return "Hello";
}
```

In the preceding example, every invocation of the `test()` method results in a message with a payload created from its return value. Each message is sent to the channel named `auditChannel`. One of the benefits of this technique is that you can avoid the duplication of the same channel name across multiple annotations. You also can provide a level of indirection between your own, potentially domain-specific, annotations and those provided by the framework.

You can also annotate the class, which lets you apply the properties of this annotation on every public method of that class, as the following example shows:

```
@Audit
static class BankingOperationsImpl implements BankingOperations {

  public String debit(String amount) {
     . . .

  }

  public String credit(String amount) {
     . . .
  }

}
```

C.1.2. XML-based Approach with the <publishing-interceptor> element

The XML-based approach lets you configure the same AOP-based message publishing functionality as a namespace-based configuration of a MessagePublishingInterceptor. It certainly has some benefits over the annotation-driven approach, since it lets you use AOP pointcut expressions, thus possibly intercepting multiple methods at once or intercepting and publishing methods to which you do not have the source code.

To configure message publishing with XML, you need only do the following two things:

- Provide configuration for MessagePublishingInterceptor by using the <publishing-interceptor> XML element.
- Provide AOP configuration to apply the MessagePublishingInterceptor to managed objects.

The following example shows how to configure a publishing-interceptor element:

```
<aop:config>
  <aop:advisor advice-ref="interceptor" pointcut="bean(testBean)" />
</aop:config>
<publishing-interceptor id="interceptor" default-channel="defaultChannel">
  <method pattern="echo" payload="'Echoing: ' + #return" channel="echoChannel">
    <header name="things" value="something"/>
  </method>
  <method pattern="repl*" payload="'Echoing: ' + #return" channel="echoChannel">
    <header name="things" expression="'something'.toUpperCase()"/>
  </method>
  <method pattern="echoDef*" payload="#return"/>
</publishing-interceptor>
```

The <publishing-interceptor> configuration looks rather similar to the annotation-based approach, and it also uses the power of the Spring Expression Language.

In the preceding example, the execution of the echo method of a testBean renders a Message with the following structure:

- The Message payload is of type String with the following content: Echoing: [value], where value is the value returned by an executed method.
- The Message has a header with a name of things and a value of something.
- The Message is sent to echoChannel.

The second method is very similar to the first. Here, every method that begins with 'repl' renders a Message with the following structure:

- The Message payload is the same as in the preceding sample.
- The Message has a header named things whose value is the result of the SpEL expression 'something'.toUpperCase().
- The Message is sent to echoChannel.

The second method, mapping the execution of any method that begins with `echoDef`, produces a `Message` with the following structure:

- The `Message` payload is the value returned by an executed method.
- Since the `channel` attribute is not provided, the `Message` is sent to the `defaultChannel` defined by the `publisher`.

For simple mapping rules you can rely on the `publisher` defaults, as the following example shows:

```
<publishing-interceptor id="anotherInterceptor"/>
```

The preceding example maps the return value of every method that matches the pointcut expression to a payload and is sent to a `default-channel`. If you do not specify the `defaultChannel` (as the preceding example does not do), the messages are sent to the global `nullChannel` (the equivalent of `/dev/null`).

Asynchronous Publishing

Publishing occurs in the same thread as your component's execution. So, by default, it is synchronous. This means that the entire message flow has to wait until the publisher's flow completes. However, developers often want the complete opposite: to use this message-publishing feature to initiate asynchronous flows. For example, you might host a service (HTTP, WS, and so on) which receives a remote request. You may want to send this request internally into a process that might take a while. However you may also want to reply to the user right away. So, instead of sending inbound requests for processing to the output channel (the conventional way), you can use 'output-channel' or a 'replyChannel' header to send a simple acknowledgment-like reply back to the caller while using the message-publisher feature to initiate a complex flow.

The service in the following example receives a complex payload (which needs to be sent further for processing), but it also needs to reply to the caller with a simple acknowledgment:

```
public String echo(Object complexPayload) {
     return "ACK";
}
```

So, instead of hooking up the complex flow to the output channel, we use the message-publishing feature instead. We configure it to create a new message, by using the input argument of the service method (shown in the preceding example), and send that to the 'localProcessChannel'. To make sure this flow is asynchronous, all we need to do is send it to any type of asynchronous channel (`ExecutorChannel` in the next example). The following example shows how to an asynchronous `publishing-interceptor`:

```
<int:service-activator  input-channel="inputChannel" output-channel="
outputChannel" ref="sampleservice"/>

<bean id="sampleservice" class="test.SampleService"/>

<aop:config>
  <aop:advisor advice-ref="interceptor" pointcut="bean(sampleservice)" />
</aop:config>

<int:publishing-interceptor id="interceptor" >
  <int:method pattern="echo" payload="#args[0]" channel="localProcessChannel">
    <int:header name="sample_header" expression="'some sample value'"/>
  </int:method>
</int:publishing-interceptor>

<int:channel id="localProcessChannel">
  <int:dispatcher task-executor="executor"/>
</int:channel>

<task:executor id="executor" pool-size="5"/>
```

Another way of handling this type of scenario is with a wire-tap. See Wire Tap.

C.1.3. Producing and Publishing Messages Based on a Scheduled Trigger

In the preceding sections, we looked at the message-publishing feature, which constructs and publishes messages as by-products of method invocations. However, in those cases, you are still responsible for invoking the method. Spring Integration 2.0 added support for scheduled message producers and publishers with the new `expression` attribute on the 'inbound-channel-adapter' element. You can schedule based on several triggers, any one of which can be configured on the 'poller' element. Currently, we support `cron`, `fixed-rate`, `fixed-delay` and any custom trigger implemented by you and referenced by the 'trigger' attribute value.

As mentioned earlier, support for scheduled producers and publishers is provided via the `<inbound-channel-adapter>` XML element. Consider the following example:

```
<int:inbound-channel-adapter id="fixedDelayProducer"
       expression="'fixedDelayTest'"
       channel="fixedDelayChannel">
    <int:poller fixed-delay="1000"/>
</int:inbound-channel-adapter>
```

The preceding example creates an inbound channel adapter that constructs a `Message`, with its payload being the result of the expression defined in the `expression` attribute. Such messages are

created and sent every time the delay specified by the `fixed-delay` attribute occurs.

The following example is similar to the preceding example, except that it uses the `fixed-rate` attribute:

```
<int:inbound-channel-adapter id="fixedRateProducer"
       expression="'fixedRateTest'"
       channel="fixedRateChannel">
    <int:poller fixed-rate="1000"/>
</int:inbound-channel-adapter>
```

The `fixed-rate` attribute lets you send messages at a fixed rate (measuring from the start time of each task).

The following example shows how you can apply a Cron trigger with a value specified in the `cron` attribute:

```
<int:inbound-channel-adapter id="cronProducer"
       expression="'cronTest'"
       channel="cronChannel">
    <int:poller cron="7 6 5 4 3 ?"/>
</int:inbound-channel-adapter>
```

The following example shows how to insert additional headers into the message:

```
<int:inbound-channel-adapter id="headerExpressionsProducer"
       expression="'headerExpressionsTest'"
       channel="headerExpressionsChannel"
       auto-startup="false">
    <int:poller fixed-delay="5000"/>
    <int:header name="foo" expression="6 * 7"/>
    <int:header name="bar" value="x"/>
</int:inbound-channel-adapter>
```

The additional message headers can take scalar values or the results of evaluating Spring expressions.

If you need to implement your own custom trigger, you can use the `trigger` attribute to provide a reference to any spring configured bean that implements the `org.springframework.scheduling.Trigger` interface. The following example shows how to do so:

```
<int:inbound-channel-adapter id="triggerRefProducer"
       expression="'triggerRefTest'" channel="triggerRefChannel">
    <int:poller trigger="customTrigger"/>
</int:inbound-channel-adapter>

<beans:bean id="customTrigger" class="o.s.scheduling.support.PeriodicTrigger">
    <beans:constructor-arg value="9999"/>
</beans:bean>
```

Appendix D: Transaction Support

This chapter covers Spring Integration's support for transactions. It covers the following topics:

- Understanding Transactions in Message flows
- Transaction Boundaries
- Transaction Synchronization
- Pseudo Transactions

D.1. Understanding Transactions in Message flows

Spring Integration exposes several hooks to address the transactional needs of your message flows. To better understand these hooks and how you can benefit from them, we must first revisit the six mechanisms that you can use to initiate message flows and see how you can address the transactional needs of these flows within each of these mechanisms.

The following six mechanisms initiate a message flow (details for each are provided throughout this manual):

- Gateway proxy: A basic messaging gateway.
- Message channel: Direct interactions with `MessageChannel` methods (for example, `channel.send(message)`).
- Message publisher: The way to initiate a message flow as the by-product of method invocations on Spring beans.
- Inbound channel adapters and gateways: The way to initiate a message flow based on connecting third-party system with the Spring Integration messaging system (for example, `[JmsMessage] → Jms Inbound Adapter[SI Message] → SI Channel`).
- Scheduler: The way to initiate a message flow based on scheduling events distributed by a pre-configured scheduler.
- Poller: Similar to the scheduler, this is the way to initiate message flow based on scheduling or interval-based events distributed by a pre-configured poller.

We can split these six mechanisms into two general categories:

- Message flows initiated by a user process: Example scenarios in this category would be invoking a gateway method or explicitly sending a `Message` to a `MessageChannel`. In other words, these message flows depend on a third party process (such as some code that you wrote) to be initiated.
- Message flows initiated by a daemon process: Example scenarios in this category include a Poller polling a message queue to initiate a new message flow with the polled message or a scheduler scheduling the process by creating a new message and initiating a message flow at a predefined time.

Clearly the gateway proxy, `MessageChannel.send(…)` and `MessagePublisher` all belong to the first category, and inbound adapters and gateways, scheduler, and poller belong to the second category.

So, how can you address transactional needs in various scenarios within each category, and is there a need for Spring Integration to provide something explicit with regard to transactions for a particular scenario? Or can you use Spring's transaction support instead?

Spring itself provides first class support for transaction management. So our goal here is not to provide something new but rather use Spring to benefit from its existing support for transactions. In other words, as a framework, we must expose hooks to Spring's transaction management functionality. However, since Spring Integration configuration is based on Spring configuration, we need not always expose these hooks, because Spring already exposes them . After all, every Spring Integration component is a Spring Bean.

With this goal in mind, we can again consider the two scenarios: messgae flows initiated by a user process and message flows initiated by a daemon.

Message flows that are initiated by a user process and configured in a Spring application context are subject to the usual transactional configuration of such processes. Therefore they need not be explicitly configured by Spring Integration to support transactions. The transaction could and should be initiated through Spring's standard transaction support. The Spring Integration message flow naturally honors the transactional semantics of the components, because it is itself configured by Spring. For example, a gateway or service activator method could be annotated with `@Transactional`, or a `TransactionInterceptor` could be defined in an XML configuration with a pointcut expression that pointa to specific methods that should be transactional. The bottom line is that you have full control over transaction configuration and boundaries in these scenarios.

However, things are a bit different when it comes to message flows initiated by a daemon process. Although configured by the developer, these flows do not directly involve a human or some other process to be initiated. These are trigger-based flows that are initiated by a trigger process (a daemon process) based on the configuration of the process. For example, we could have a scheduler initiate a message flow every Friday night. We can also configure a trigger that initiates a message flow every second and so on. As a result, we need a way to let these trigger-based processes know of our intention to make the resulting message flows be transactional, so that a Transaction context can be created whenever a new message flow is initiated. In other words, we need to expose some transaction configuration, but only enough to delegate to the transaction support already provided by Spring (as we do in other scenarios).

D.1.1. Poller Transaction Support

Spring Integration provides transactional support for pollers. Pollers are a special type of component because, within a poller task, we can call `receive()` against a resource that is itself transactional, thus including the `receive()` call in the the boundaries of the transaction, which lets it be rolled back in case of a task failure. If we were to add the same support for channels, the added transactions would affect all downstream components starting with the `send()` call. That provides a rather wide scope for transaction demarcation without any strong reason, especially when Spring already provides several ways to address the transactional needs of any component downstream. However the `receive()` method being included in a transaction boundary is the "strong reason" for pollers.

Any time you configure a Poller, you can provide transactional configuration by using the `transactional` child element and its attributes,as the following example shows:

```
<int:poller max-messages-per-poll="1" fixed-rate="1000">
    <transactional transaction-manager="txManager"
                   isolation="DEFAULT"
                   propagation="REQUIRED"
                   read-only="true"
                   timeout="1000"/>
</poller>
```

The preceding configuration looks similar to a native Spring transaction configuration. You must still provide a reference to a transaction manager and either specify transaction attributes or rely on defaults (for example, if the 'transaction-manager' attribute is not specified, it defaults to the bean named 'transactionManager'). Internally, the process is wrapped in Spring's native transaction, where `TransactionInterceptor` is responsible for handling transactions. For more information on how to configure a transaction manager, the types of transaction managers (such as JTA, Datasource, and others), and other details related to transaction configuration, see the Spring Framework Reference Guide.

With the preceding configuration, all message flows initiated by this poller are transactional. For more information and details on a poller's transactional configuration, see Polling and Transactions.

Along with transactions, you might need to address several more cross-cutting concerns when you run a poller. To help with that, the poller element accepts an `<advice-chain>` child element, which lets you define a custom chain of advice instances to be applied on the Poller. (See Pollable Message Source for more details.) In Spring Integration 2.0, the Poller went through a refactoring effort and now uses a proxy mechanism to address transactional concerns as well as other cross-cutting concerns. One of the significant changes evolving from this effort is that we made the `<transactional>` and `<advice-chain>` elements be mutually exclusive. The rationale behind this is that, if you need more than one advice and one of them is Transaction advice, you can include it in the `<advice-chain>` with the same convenience as before but with much more control, since you now have an option to position the advice in the desired order. The following example shows how to do so:

```
<int:poller max-messages-per-poll="1" fixed-rate="10000">
  <advice-chain>
    <ref bean="txAdvice"/>
    <ref bean="someAotherAdviceBean" />
    <beans:bean class="foo.bar.SampleAdvice"/>
  </advice-chain>
</poller>

<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

The preceding example shows a basic XML-based configuration of Spring Transaction advice (`txAdvice`) and included it within the `<advice-chain>` defined by the Poller. If you need to address only the transactional concerns of the poller, you can still use the `<transactional>` element as a convenience.

D.2. Transaction Boundaries

Another important factor is the boundaries of Transactions within a Message flow. When a transaction is started, the transaction context is bound to the current thread. So regardless of how many endpoints and channels you have in your Message flow your transaction context will be preserved as long as you are ensuring that the flow continues on the same thread. As soon as you break it by introducing a *Pollable Channel* or *Executor Channel* or initiate a new thread manually in some service, the Transactional boundary will be broken as well. Essentially the Transaction will END right there, and if a successful handoff has transpired between the threads, the flow would be considered a success and a COMMIT signal would be sent even though the flow will continue and might still result in an Exception somewhere downstream. If such a flow were synchronous, that Exception could be thrown back to the initiator of the Message flow who is also the initiator of the transactional context and the transaction would result in a ROLLBACK. The middle ground is to use transactional channels at any point where a thread boundary is being broken. For example, you can use a Queue-backed Channel that delegates to a transactional MessageStore strategy, or you could use a JMS-backed channel.

D.3. Transaction Synchronization

In some environments, it help to synchronize operations with a transaction that encompasses the entire flow. For example, consider a `<file:inbound-channel-adapter/>` at the start of a flow that performs a number of database updates. If the transaction commits, we might want to move the file to a `success` directory, while we might want to move it to a `failure` directory if the transaction rolls back.

Spring Integration 2.2 introduced the capability of synchronizing these operations with a transaction. In addition, you can configure a `PseudoTransactionManager` if you do not have a 'real' transaction but still want to perform different actions on success or failure. For more information, see Pseudo Transactions.

The following listing shows the key strategy interfaces for this feature:

```
public interface TransactionSynchronizationFactory {

    TransactionSynchronization create(Object key);
}

public interface TransactionSynchronizationProcessor {

    void processBeforeCommit(IntegrationResourceHolder holder);

    void processAfterCommit(IntegrationResourceHolder holder);

    void processAfterRollback(IntegrationResourceHolder holder);

}
```

The factory is responsible for creating a `TransactionSynchronization` object. You can implement your own or use the one provided by the framework: `DefaultTransactionSynchronizationFactory`. This implementation returns a `TransactionSynchronization` that delegates to a default implementation of `TransactionSynchronizationProcessor`: `ExpressionEvaluatingTransactionSynchronizationProcessor`. This processor supports three SpEL expressions: `beforeCommitExpression`, `afterCommitExpression`, and `afterRollbackExpression`.

These actions should be self-explanatory to those familiar with transactions. In each case, the `#root` variable is the original `Message`. In some cases, other SpEL variables are made available, depending on the `MessageSource` being polled by the poller. For example, the `MongoDbMessageSource` provides the `#mongoTemplate` variable, which references the message source's `MongoTemplate`. Similarly, the `RedisStoreMessageSource` provides the `#store` variable, which references the `RedisStore` created by the poll.

To enable the feature for a particular poller, you can provide a reference to the `TransactionSynchronizationFactory` on the poller's `<transactional/>` element by using the `synchronization-factory` attribute.

Starting with version 5.0, Spring Integration provides `PassThroughTransactionSynchronizationFactory`, which is applied by default to polling endpoints when no `TransactionSynchronizationFactory` is configured but an advice of type `TransactionInterceptor` exists in the advice chain. When using any out-of-the-box `TransactionSynchronizationFactory` implementation, polling endpoints bind a polled message to the current transactional context and provide it as a `failedMessage` in a `MessagingException` if an exception is thrown after the transaction advice. When using a custom transaction advice that does

not implement `TransactionInterceptor`, you can explicitly configure a `PassThroughTransactionSynchronizationFactory` to achieve this behavior. In either case, the `MessagingException` becomes the payload of the `ErrorMessage` that is sent to the `errorChannel`, and the cause is the raw exception thrown by the advice. Previously, the `ErrorMessage` had a payload that was the raw exception thrown by the advice and did not provide a reference to the `failedMessage` information, making it difficult to determine the reasons for the transaction commit problem.

To simplify configuration of these components, Spring Integration provides namespace support for the default factory. The following example shows how to use the namespace to configure a file inbound channel adapter:

```
<int-file:inbound-channel-adapter id="inputDirPoller"
    channel="someChannel"
    directory="/foo/bar"
    filter="filter"
    comparator="testComparator">
    <int:poller fixed-rate="5000">
        <int:transactional transaction-manager="transactionManager"
synchronization-factory="syncFactory" />
    </int:poller>
</int-file:inbound-channel-adapter>

<int:transaction-synchronization-factory id="syncFactory">
    <int:after-commit expression="payload.renameTo(new java.io.File('/success/' +
payload.name))"
        channel="committedChannel" />
    <int:after-rollback expression="payload.renameTo(new java.io.File('/failed/' +
payload.name))"
        channel="rolledBackChannel" />
</int:transaction-synchronization-factory>
```

The result of the SpEL evaluation is sent as the payload to either `committedChannel` or `rolledBackChannel` (in this case, this would be `Boolean.TRUE` or `Boolean.FALSE` — the result of the `java.io.File.renameTo()` method call).

If you wish to send the entire payload for further Spring Integration processing, use the 'payload' expression.

It is important to understand that this synchronizes the actions with a transaction. It does not make a resource that is not inherently transactional actually be transactional. Instead, the transaction (be it JDBC or otherwise) is started before the poll and either committed or rolled back when the flow completes, followed by the synchronized action.

If you provide a custom `TransactionSynchronizationFactory`, it is responsible for creating a resource synchronization that causes the bound resource to be unbound automatically when the transaction completes. The default `TransactionSynchronizationFactory` does so by returning a subclass of `ResourceHolderSynchronization`, with the default `shouldUnbindAtCompletion()` returning `true`.

In addition to the `after-commit` and `after-rollback` expressions, `before-commit` is also supported. In that case, if the evaluation (or downstream processing) throws an exception, the transaction is rolled back instead of being committed.

D.4. Pseudo Transactions

After reading the Transaction Synchronization section, you might think it would be useful to take these 'success' or 'failure' actions when a flow completes, even if there is no “real” transactional resources (such as JDBC) downstream of the poller. For example, consider a “<file:inbound-channel-adapter/>” followed by an “<ftp:outbout-channel-adapter/>”. Neither of these components is transactional, but we might want to move the input file to different directories, based on the success or failure of the FTP transfer.

To provide this functionality, the framework provides a `PseudoTransactionManager`, enabling the above configuration even when there is no real transactional resource involved. If the flow completes normally, the `beforeCommit` and `afterCommit` synchronizations are called. On failure, the `afterRollback` synchronization is called. Because it is not a real transaction, no actual commit or rollback occurs. The pseudo transaction is a vehicle used to enable the synchronization features.

To use a `PseudoTransactionManager`, you can define it as a <bean/>, in the same way you would configure a real transaction manager. The following example shows how to do so:

```
<bean id="transactionManager" class="o.s.i.transaction.PseudoTransactionManager"
/>
```

D.5. Reactive Transactions

Starting with version 5.3, a `ReactiveTransactionManager` can also be used together with a `TransactionInterceptor` advice for endpoints returning a reactive type. This includes `MessageSource` and `ReactiveMessageHandler` implementations (e.g. `ReactiveMongoDbMessageSource`) which produce a message with a `Flux` or `Mono` payload. All other reply producing message handler implementations can rely on a `ReactiveTransactionManager` when their reply payload is also some reactive type.

Appendix E: Security in Spring Integration

Security is one of the important functions in any modern enterprise (or cloud) application. Moreover, it is critical for distributed systems, such as those built on Enterprise Integration Patterns. Messaging independence and loose coupling let target systems communicate with each other with any type of data in the message's `payload`. We can either trust all those messages or secure our service against "infecting" messages.

Spring Integration, together with Spring Security, provides a simple and comprehensive way to secure message channels, as well as other part of the integration solution.

You need to include this dependency into your project:

Maven

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-security</artifactId>
    <version>5.3.0.RC1</version>
</dependency>
```

Gradle

```
compile "org.springframework.integration:spring-integration-security:5.3.0.RC1"
```

E.1. Securing channels

Spring Integration provides the `ChannelSecurityInterceptor` interceptor, which extends `AbstractSecurityInterceptor` and intercepts send and receive calls on the channel. Access decisions are then made with reference to a `ChannelSecurityMetadataSource`, which provides the metadata that describes the send and receive access policies for certain channels. The interceptor requires that a valid `SecurityContext` has been established by authenticating with Spring Security. See the Spring Security Reference Guide for details.

Spring Integration provides Namespace support to allow easy configuration of security constraints. This support consists of the secured channels tag, which allows definition of one or more channel name patterns in conjunction with a definition of the security configuration for send and receive. The pattern is a `java.util.regexp.Pattern`.

The following example shows how to configure a bean that includes security and how to set up policies with patterns:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:int="http://www.springframework.org/schema/integration"
   xmlns:int-security="http://www.springframework.org/schema/integration/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:security="http://www.springframework.org/schema/security"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
      https://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/security
      https://www.springframework.org/schema/security/spring-security.xsd
      http://www.springframework.org/schema/integration
      https://www.springframework.org/schema/integration/spring-integration.xsd
      http://www.springframework.org/schema/integration/security
      https://www.springframework.org/schema/integration/security/spring-
integration-security.xsd">

<int-security:secured-channels>
    <int-security:access-policy pattern="admin.*" send-access="ROLE_ADMIN"/>
    <int-security:access-policy pattern="user.*" receive-access="ROLE_USER"/>
</int-security:secured-channels>
```

By default, the `secured-channels` namespace element expects a bean named `authenticationManager` (which implements `AuthenticationManager`) and a bean named `accessDecisionManager` (which implements `AccessDecisionManager`). Where this is not the case, references to the appropriate beans can be configured as attributes of the `secured-channels` element, as the following example shows:

```
<int-security:secured-channels access-decision-manager=
"customAccessDecisionManager"
                               authentication-manager="customAuthenticationManager
">
    <int-security:access-policy pattern="admin.*" send-access="ROLE_ADMIN"/>
    <int-security:access-policy pattern="user.*" receive-access="ROLE_USER"/>
</int-security:secured-channels>
```

Starting with version 4.2, the `@SecuredChannel` annotation is available for Java configuration in `@Configuration` classes.

The following example shows the Java equivalent of the preceding XML examples:

```
@Configuration
@EnableIntegration
public class ContextConfiguration {

    @Bean
    @SecuredChannel(interceptor = "channelSecurityInterceptor", sendAccess =
"ROLE_ADMIN")
    public SubscribableChannel adminChannel() {
        return new DirectChannel();
    }

    @Bean
    @SecuredChannel(interceptor = "channelSecurityInterceptor", receiveAccess =
"ROLE_USER")
    public SubscribableChannel userChannel() {
        return new DirectChannel();
    }

    @Bean
    public ChannelSecurityInterceptor channelSecurityInterceptor(
            AuthenticationManager authenticationManager,
            AccessDecisionManager accessDecisionManager) {
        ChannelSecurityInterceptor channelSecurityInterceptor = new
ChannelSecurityInterceptor();
        channelSecurityInterceptor.setAuthenticationManager(authenticationManager
);
        channelSecurityInterceptor.setAccessDecisionManager(accessDecisionManager
);
        return channelSecurityInterceptor;
    }

}
```

E.2. Security Context Propagation

To be sure that our interaction with the application is secure, according to its security system rules, we should supply some security context with an authentication (principal) object. The Spring Security project provides a flexible, canonical mechanism to authenticate our application clients over HTTP, WebSocket, or SOAP protocols (as can be done for any other integration protocol with a simple Spring Security extension). It also provides a `SecurityContext` for further authorization checks on the application objects, such as message channels. By default, the `SecurityContext` is tied to the execution state of the current `Thread` by using the (`ThreadLocalSecurityContextHolderStrategy`). It is accessed by an AOP (Aspect-oriented Programming) interceptor on secured methods to check (for example) whether that `principal` of the invocation has sufficient permissions to call that method. This works well with the current thread. Often, though, processing logic can be performed on another thread, on several threads, or even on external systems.

Standard thread-bound behavior is easy to configure if our application is built on the Spring Integration components and its message channels. In this case, the secured objects can be any service activator or transformer, secured with a `MethodSecurityInterceptor` in their `<request-handler-advice-chain>` (see Adding Behavior to Endpoints) or even `MessageChannel` (see Securing channels, earlier). When using `DirectChannel` communication, the `SecurityContext` is automatically available, because the downstream flow runs on the current thread. However, in the cases of the `QueueChannel`, `ExecutorChannel`, and `PublishSubscribeChannel` with an `Executor`, messages are transferred from one thread to another (or several) by the nature of those channels. In order to support such scenarios, we have two choices:

- Transfer an `Authentication` object within the message headers and extract and authenticate it on the other side before secured object access.
- Propagate the `SecurityContext` to the thread that receives the transferred message.

Version 4.2 introduced `SecurityContext` propagation. It is implemented as a `SecurityContextPropagationChannelInterceptor`, which you can add to any `MessageChannel` or configure as a `@GlobalChannelInterceptor`. The logic of this interceptor is based on the `SecurityContext` extraction from the current thread (from the `preSend()` method) and its populating to another thread from the `postReceive()` (`beforeHandle()`) method. Actually, this interceptor is an extension of the more generic `ThreadStatePropagationChannelInterceptor`, which wraps the message to send with the state to propagate in an internal `Message<?>` extension (`MessageWithThreadState<S>`) on one side and extracts the original message and the state to propagate on the other side. You can extend the `ThreadStatePropagationChannelInterceptor` for any context propagation use case, and `SecurityContextPropagationChannelInterceptor` is a good example of doing so.

The logic of the `ThreadStatePropagationChannelInterceptor` is based on message modification (it returns an internal `MessageWithThreadState` object to send). Consequently, you should be careful when combining this interceptor with any other that can also modify messages (for example, through the `MessageBuilder.withPayload(…)…build()`). The state to propagate may be lost. In most cases, to overcome the issue, you can order the interceptors for the channel and ensure the `ThreadStatePropagationChannelInterceptor` is the last one in the stack.

Propagation and population of `SecurityContext` is just one half of the work. Since the message is not an owner of the threads in the message flow and we should be sure that we are secure against any incoming messages, we have to clean up the `SecurityContext` from `ThreadLocal`. The `SecurityContextPropagationChannelInterceptor` provides the `afterMessageHandled()` interceptor method implementation. It cleans up operation by freeing the thread at the end of invocation from that propagated principal. This means that, when the thread that processes the handed-off message finishes processing the message (successful or otherwise), the context is cleared so that it cannot inadvertently be used when processing another message.

When working with an asynchronous gateway, you should use an appropriate `AbstractDelegatingSecurityContextSupport` implementation from Spring Security Concurrency Support, to let security context propagation be ensured over gateway invocation. The following example shows how to do so:

```
@Configuration
@EnableIntegration
@IntegrationComponentScan
public class ContextConfiguration {

    @Bean
    public AsyncTaskExecutor securityContextExecutor() {
        return new DelegatingSecurityContextAsyncTaskExecutor(
                            new SimpleAsyncTaskExecutor());
    }

}

...

@MessagingGateway(asyncExecutor = "securityContextExecutor")
public interface SecuredGateway {

    @Gateway(requestChannel = "queueChannel")
    Future<String> send(String payload);

}
```

Appendix F: Configuration

Spring Integration offers a number of configuration options. Which option you choose depends upon your particular needs and at what level you prefer to work. As with the Spring framework in general, you can mix and match the various techniques to suit the problem at hand. For example, you can choose the XSD-based namespace for the majority of configuration and combine it with a handful of objects that you configure with annotations. As much as possible, the two provide consistent naming. The XML elements defined by the XSD schema match the names of the annotations, and the attributes of those XML elements match the names of annotation properties. You can also use the API directly, but we expect most developers to choose one of the higher-level options or a combination of the namespace-based and annotation-driven configuration.

F.1. Namespace Support

You can configure Spring Integration components with XML elements that map directly to the terminology and concepts of enterprise integration. In many cases, the element names match those of the *Enterprise Integration Patterns* book.

To enable Spring Integration's core namespace support within your Spring configuration files, add the following namespace reference and schema mapping in your top-level 'beans' element:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:int="http://www.springframework.org/schema/integration"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           https://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/integration
           https://www.springframework.org/schema/integration/spring-
integration.xsd">
```

(We have emphasized the lines that are particular to Spring Integration.)

You can choose any name after "xmlns:". We use `int` (short for Integration) for clarity, but you might prefer another abbreviation. On the other hand, if you use an XML editor or IDE support, the availability of auto-completion may convince you to keep the longer name for clarity. Alternatively, you can create configuration files that use the Spring Integration schema as the primary namespace, as the following example shows:

```
<beans:beans xmlns="http://www.springframework.org/schema/integration"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:beans="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           https://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/integration
           https://www.springframework.org/schema/integration/spring-
integration.xsd">
```

(We have emphasized the lines that are particular to Spring Integration.)

When using this alternative, no prefix is necessary for the Spring Integration elements. On the other hand, if you define a generic Spring bean within the same configuration file, the bean element requires a prefix (`<beans:bean …/>`). Since it is generally a good idea to modularize the configuration files themselves (based on responsibility or architectural layer), you may find it appropriate to use the latter approach in the integration-focused configuration files, since generic beans are seldom necessary within those files. For the purposes of this documentation, we assume the integration namespace is the primary.

Spring Integration provides many other namespaces. In fact, each adapter type (JMS, file, and so on) that provides namespace support defines its elements within a separate schema. In order to use these elements, add the necessary namespaces with an `xmlns` entry and the corresponding `schemaLocation` mapping. For example, the following root element shows several of these namespace declarations:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-file="http://www.springframework.org/schema/integration/file"
  xmlns:int-jms="http://www.springframework.org/schema/integration/jms"
  xmlns:int-mail="http://www.springframework.org/schema/integration/mail"
  xmlns:int-rmi="http://www.springframework.org/schema/integration/rmi"
  xmlns:int-ws="http://www.springframework.org/schema/integration/ws"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    https://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/file
    https://www.springframework.org/schema/integration/file/spring-integration-
file.xsd
    http://www.springframework.org/schema/integration/jms
    https://www.springframework.org/schema/integration/jms/spring-integration-
jms.xsd
    http://www.springframework.org/schema/integration/mail
    https://www.springframework.org/schema/integration/mail/spring-integration-
mail.xsd
    http://www.springframework.org/schema/integration/rmi
    https://www.springframework.org/schema/integration/rmi/spring-integration-
rmi.xsd
    http://www.springframework.org/schema/integration/ws
    https://www.springframework.org/schema/integration/ws/spring-integration-
ws.xsd">
 ...
</beans>
```

This reference manual provides specific examples of the various elements in their corresponding chapters. Here, the main thing to recognize is the consistency of the naming for each namespace URI and schema location.

F.2. Configuring the Task Scheduler

In Spring Integration, the `ApplicationContext` plays the central role of a message bus, and you need to consider only a couple of configuration options. First, you may want to control the central `TaskScheduler` instance. You can do so by providing a single bean named `taskScheduler`. This is also defined as a constant, as follows:

```
IntegrationContextUtils.TASK_SCHEDULER_BEAN_NAME
```

By default, Spring Integration relies on an instance of `ThreadPoolTaskScheduler`, as described in the Task Execution and Scheduling section of the Spring Framework reference manual. That default `TaskScheduler` starts up automatically with a pool of ten threads, but see Global Properties. If you provide your own `TaskScheduler` instance instead, you can set the 'autoStartup' property to `false` or provide your own pool size value.

When polling consumers provide an explicit task executor reference in their configuration, the invocation of the handler methods happens within that executor's thread pool and not the main scheduler pool. However, when no task executor is provided for an endpoint's poller, it is invoked by one of the main scheduler's threads.

Do not run long-running tasks on poller threads. Use a task executor instead. If you have a lot of polling endpoints, you can cause thread starvation, unless you increase the pool size. Also, polling consumers have a default `receiveTimeout` of one second. Since the poller thread blocks for this time, we recommend that you use a task executor when many such endpoints exist, again to avoid starvation. Alternatively, you can reduce the `receiveTimeout`.

An endpoint is a Polling Consumer if its input channel is one of the queue-based (that is, pollable) channels. Event-driven consumers are those having input channels that have dispatchers instead of queues (in other words, they are subscribable). Such endpoints have no poller configuration, since their handlers are invoked directly.

When running in a JEE container, you may need to use Spring's `TimerManagerTaskScheduler`, as described here, instead of the default `taskScheduler`. To do so, define a bean with the appropriate JNDI name for your environment, as the following example shows:

```
<bean id="taskScheduler" class=
"o.s.scheduling.commonj.TimerManagerTaskScheduler">
    <property name="timerManagerName" value="tm/MyTimerManager" />
    <property name="resourceRef" value="true" />
</bean>
```

The next section describes what happens if exceptions occur within the asynchronous invocations.

F.3. Global Properties

Certain global framework properties can be overridden by providing a properties file on the classpath.

The default properties can be found in `/META-INF/spring.integration.default.properties` in the `spring-integration-core` jar. You can see them on GitHub here. The following listing shows the default values:

```
spring.integration.channels.autoCreate=true ①
spring.integration.channels.maxUnicastSubscribers=0x7fffffff ②
spring.integration.channels.maxBroadcastSubscribers=0x7fffffff ③
spring.integration.taskScheduler.poolSize=10 ④
spring.integration.messagingTemplate.throwExceptionOnLateReply=false ⑤
spring.integration.readOnly.headers= ⑥
spring.integration.endpoints.noAutoStartup= ⑦
spring.integration.postProcessDynamicBeans=false ⑧
```

① When true, `input-channel` instances are automatically declared as `DirectChannel` instances when not explicitly found in the application context.

② Sets the default number of subscribers allowed on, for example, a `DirectChannel`. It can be used to avoid inadvertently subscribing multiple endpoints to the same channel. You can override it on individual channels by setting the `max-subscribers` attribute.

③ This property provides the default number of subscribers allowed on, for example, a `PublishSubscribeChannel`. It can be used to avoid inadvertently subscribing more than the expected number of endpoints to the same channel. You can override it on individual channels by setting the `max-subscribers` attribute.

④ The number of threads available in the default `taskScheduler` bean. See Configuring the Task Scheduler.

⑤ When `true`, messages that arrive at a gateway reply channel throw an exception when the gateway is not expecting a reply (because the sending thread has timed out or already received a reply).

⑥ A comma-separated list of message header names that should not be populated into `Message` instances during a header copying operation. The list is used by the `DefaultMessageBuilderFactory` bean and propagated to the `IntegrationMessageHeaderAccessor` instances (see `MessageHeaderAccessor` API) used to build messages via `MessageBuilder` (see The `MessageBuilder` Helper Class). By default, only `MessageHeaders.ID` and `MessageHeaders.TIMESTAMP` are not copied during message building. Since version 4.3.2.

⑦ A comma-separated list of `AbstractEndpoint` bean names patterns (`xxx*`, `xxx`, `*xxx` or `xxx*yyy`) that should not be started automatically during application startup. You can manually start these endpoints later by their bean name through a `Control Bus` (see Control Bus), by their role with the `SmartLifecycleRoleController` (see Endpoint Roles), or by `Lifecycle` bean injection. You can explicitly override the effect of this global property by specifying `auto-startup` XML annotation or the `autoStartup` annotation attribute or by calling `AbstractEndpoint.setAutoStartup()` in the bean definition. Since version 4.3.12.

⑧ A boolean flag to indicate that `BeanPostProcessor` instances should post-process beans registered at runtime (for example, message channels created by `IntegrationFlowContext` can be supplied with global channel interceptors). Since version 4.3.15.

These properties can be overridden by adding a `/META-INF/spring.integration.properties` file to the classpath. You need not provide all the properties — only those that you want to override.

Starting with version 5.1, all the merged global properties are printed in the logs after application

context startup when a DEBUG logic level is turned on for the org.springframework.integration category. The output looks like this:

```
Spring Integration global properties:

spring.integration.endpoints.noAutoStartup=fooService*
spring.integration.taskScheduler.poolSize=20
spring.integration.channels.maxUnicastSubscribers=0x7fffffff
spring.integration.channels.autoCreate=true
spring.integration.channels.maxBroadcastSubscribers=0x7fffffff
spring.integration.readOnly.headers=
spring.integration.messagingTemplate.throwExceptionOnLateReply=true
```

F.4. Annotation Support

In addition to the XML namespace support for configuring message endpoints, you can also use annotations. First, Spring Integration provides the class-level @MessageEndpoint as a stereotype annotation, meaning that it is itself annotated with Spring's @Component annotation and is therefore automatically recognized as a bean definition by Spring's component scanning.

Even more important are the various method-level annotations. They indicate that the annotated method is capable of handling a message. The following example demonstrates both class-level and method-level annotations:

```
@MessageEndpoint
public class FooService {

    @ServiceActivator
    public void processMessage(Message message) {
        ...
    }
}
```

Exactly what it means for the method to "handle" the Message depends on the particular annotation. Annotations available in Spring Integration include:

- @Aggregator (see Aggregator)
- @Filter (see Filter)
- @Router (see Routers)
- @ServiceActivator (see Service Activator)
- @Splitter (see Splitter)
- @Transformer (see Transformer)

- `@InboundChannelAdapter` (see Channel Adapter)
- `@BridgeFrom` (see Configuring a Bridge with Java Configuration)
- `@BridgeTo` (see Configuring a Bridge with Java Configuration)
- `@MessagingGateway` (see Messaging Gateways)
- `@IntegrationComponentScan` (see Configuration and `@EnableIntegration`)

If you use XML configuration in combination with annotations, the `@MessageEndpoint` annotation is not required. If you want to configure a POJO reference from the `ref` attribute of a `<service-activator/>` element, you can provide only the method-level annotations. In that case, the annotation prevents ambiguity even when no method-level attribute exists on the `<service-activator/>` element.

In most cases, the annotated handler method should not require the `Message` type as its parameter. Instead, the method parameter type can match the message's payload type, as the following example shows:

```
public class ThingService {

    @ServiceActivator
    public void bar(Thing thing) {
        ...
    }

}
```

When the method parameter should be mapped from a value in the `MessageHeaders`, another option is to use the parameter-level `@Header` annotation. In general, methods annotated with the Spring Integration annotations can accept the `Message` itself, the message payload, or a header value (with `@Header`) as the parameter. In fact, the method can accept a combination, as the following example shows:

```
public class ThingService {

    @ServiceActivator
    public void otherThing(String payload, @Header("x") int valueX, @Header("y")
int valueY) {
        ...
    }

}
```

You can also use the `@Headers` annotation to provide all of the message headers as a `Map`, as the following example shows:

```
public class ThingService {

    @ServiceActivator
    public void otherThing(String payload, @Headers Map<String, Object> headerMap)
{
        ...
    }

}
```

The value of the annotation can also be a SpEL expression (for example, `someHeader.toUpperCase()`), which is useful when you wish to manipulate the header value before injecting it. It also provides an optional `required` property, which specifies whether the attribute value must be available within the headers. The default value for the `required` property is `true`.

For several of these annotations, when a message-handling method returns a non-null value, the endpoint tries to send a reply. This is consistent across both configuration options (namespace and annotations) in that such an endpoint's output channel is used (if available), and the `REPLY_CHANNEL` message header value is used as a fallback.

The combination of output channels on endpoints and the reply channel message header enables a pipeline approach, where multiple components have an output channel and the final component allows the reply message to be forwarded to the reply channel (as specified in the original request message). In other words, the final component depends on the information provided by the original sender and can dynamically support any number of clients as a result. This is an example of the return address pattern.

In addition to the examples shown here, these annotations also support the `inputChannel` and `outputChannel` properties, as the following example shows:

```
@Service
public class ThingService {

    @ServiceActivator(inputChannel="input", outputChannel="output")
    public void otherThing(String payload, @Headers Map<String, Object> headerMap)
{
        ...
    }

}
```

The processing of these annotations creates the same beans as the corresponding XML components — `AbstractEndpoint` instances and `MessageHandler` instances (or `MessageSource` instances for the inbound channel adapter). See Annotations on `@Bean` Methods. The bean names are generated from the following pattern: `[componentName].[methodName].[decapitalizedAnnotationClassShortName]`. In the preceding example the bean name is `thingService.otherThing.serviceActivator` for the `AbstractEndpoint` and the same name with an additional `.handler` (`.source`) suffix for the `MessageHandler` (`MessageSource`) bean. Such a name can be customized using an `@EndpointId` annotation alongside with these messaging annotations. The `MessageHandler` instances (`MessageSource` instances) are also eligible to be tracked by the message history.

Starting with version 4.0, all messaging annotations provide `SmartLifecycle` options (`autoStartup` and `phase`) to allow endpoint lifecycle control on application context initialization. They default to `true` and `0`, respectively. To change the state of an endpoint (such as ` start()` or `stop()`), you can obtain a reference to the endpoint bean by using the `BeanFactory` (or autowiring) and invoke the methods. Alternatively, you can send a command message to the `Control Bus` (see Control Bus). For these purposes, you should use the `beanName` mentioned earlier in the preceding paragraph.

Channels automatically created after parsing the mentioned annotations (when no specific channel bean is configured), and the corresponding consumer endpoints, are declared as beans near the end of the context initialization. These beans **can** be autowired in other services, but they have to be marked with the `@Lazy` annotation because the definitions, typically, won't yet be available during normal autowiring processing.

```
@Autowired
@Lazy
@Qualifier("someChannel")
MessageChannel someChannel;
...

@Bean
Thing1 dependsOnSPCA(@Qualifier("someInboundAdapter") @Lazy
SourcePollingChannelAdapter someInboundAdapter) {
    ...
}
```

F.4.1. Using the `@Poller` Annotation

Before Spring Integration 4.0, messaging annotations required that the `inputChannel` be a reference to a `SubscribableChannel`. For `PollableChannel` instances, an `<int:bridge/>` element was needed to configure an `<int:poller/>` and make the composite endpoint be a `PollingConsumer`. Version 4.0 introduced the `@Poller` annotation to allow the configuration of `poller` attributes directly on the messaging annotations, as the following example shows:

```
public class AnnotationService {

    @Transformer(inputChannel = "input", outputChannel = "output",
        poller = @Poller(maxMessagesPerPoll = "${poller.maxMessagesPerPoll}",
fixedDelay = "${poller.fixedDelay}"))
    public String handle(String payload) {
        ...
    }
}
```

The `@Poller` annotation provides only simple `PollerMetadata` options. You can configure the `@Poller` annotation's attributes (`maxMessagesPerPoll`, `fixedDelay`, `fixedRate`, and `cron`) with property placeholders. Also, starting with version 5.1, the `receiveTimeout` option for `PollingConsumer` s is also provided. If it is necessary to provide more polling options (for example, `transaction`, `advice-chain`, `error-handler`, and others), you should configure the `PollerMetadata` as a generic bean and use its bean name as the `@Poller` 's `value` attribute. In this case, no other attributes are allowed (they must

be specified on the `PollerMetadata` bean). Note, if `inputChannel` is a `PollableChannel` and no `@Poller` is configured, the default `PollerMetadata` is used (if it is present in the application context). To declare the default poller by using a `@Configuration` annotation, use code similar to the following example:

```
@Bean(name = PollerMetadata.DEFAULT_POLLER)
public PollerMetadata defaultPoller() {
    PollerMetadata pollerMetadata = new PollerMetadata();
    pollerMetadata.setTrigger(new PeriodicTrigger(10));
    return pollerMetadata;
}
```

The following example shows how to use the default poller:

```
public class AnnotationService {

    @Transformer(inputChannel = "aPollableChannel", outputChannel = "output")
    public String handle(String payload) {
        ...
    }
}
```

The following example shows how to use a named poller:

```
@Bean
public PollerMetadata myPoller() {
    PollerMetadata pollerMetadata = new PollerMetadata();
    pollerMetadata.setTrigger(new PeriodicTrigger(1000));
    return pollerMetadata;
}
```

The following example shows an endpoint that uses the default poller:

```
public class AnnotationService {

    @Transformer(inputChannel = "aPollableChannel", outputChannel = "output"
                            poller = @Poller("myPoller"))
    public String handle(String payload) {
         ...
    }
}
```

Starting with version 4.3.3, the `@Poller` annotation has the `errorChannel` attribute for easier configuration of the underlying `MessagePublishingErrorHandler`. This attribute plays the same role as `error-channel` in the `<poller>` XML component. See Endpoint Namespace Support for more information.

F.4.2. Using the `@InboundChannelAdapter` Annotation

Version 4.0 introduced the `@InboundChannelAdapter` method-level annotation. It produces a `SourcePollingChannelAdapter` integration component based on a `MethodInvokingMessageSource` for the annotated method. This annotation is an analogue of the `<int:inbound-channel-adapter>` XML component and has the same restrictions: The method cannot have parameters, and the return type must not be `void`. It has two attributes: `value` (the required `MessageChannel` bean name) and `poller` (an optional `@Poller` annotation, as described earlier). If you need to provide some `MessageHeaders`, use a `Message<?>` return type and use a `MessageBuilder` to build the `Message<?>`. Using a `MessageBuilder` lets you configure the `MessageHeaders`. The following example shows how to use an `@InboundChannelAdapter` annotation:

```
@InboundChannelAdapter("counterChannel")
public Integer count() {
    return this.counter.incrementAndGet();
}

@InboundChannelAdapter(value = "fooChannel", poller = @Poller(fixed-rate = "5000"
))
public String foo() {
    return "foo";
}
```

Version 4.3 introduced the `channel` alias for the `value` annotation attribute, to provide better source code readability. Also, the target `MessageChannel` bean is resolved in the `SourcePollingChannelAdapter` by the provided name (set by the `outputChannelName` option) on the first `receive()` call, not during the initialization phase. It allows "late binding" logic: The target `MessageChannel` bean from the consumer perspective is created and registered a bit later than the `@InboundChannelAdapter` parsing phase.

The first example requires that the default poller has been declared elsewhere in the application context.

Using the `@MessagingGateway` Annotation

See `@MessagingGateway` Annotation.

F.4.3. Using the `@IntegrationComponentScan` Annotation

The standard Spring Framework `@ComponentScan` annotation does not scan interfaces for stereotype `@Component` annotations. To overcome this limitation and allow the configuration of `@MessagingGateway` (see `@MessagingGateway` Annotation), we introduced the `@IntegrationComponentScan` mechanism. This annotation must be placed with a `@Configuration` annotation and be customized to define its scanning options, such as `basePackages` and `basePackageClasses`. In this case, all discovered interfaces annotated with `@MessagingGateway` are parsed and registered as `GatewayProxyFactoryBean` instances. All other class-based components are parsed by the standard `@ComponentScan`.

F.5. Messaging Meta-Annotations

Starting with version 4.0, all messaging annotations can be configured as meta-annotations and all user-defined messaging annotations can define the same attributes to override their default values. In addition, meta-annotations can be configured hierarchically, as the following example shows:

```
@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@ServiceActivator(inputChannel = "annInput", outputChannel = "annOutput")
public @interface MyServiceActivator {

    String[] adviceChain = { "annAdvice" };
}

@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@MyServiceActivator
public @interface MyServiceActivator1 {

    String inputChannel();

    String outputChannel();
}
...

@MyServiceActivator1(inputChannel = "inputChannel", outputChannel = "
outputChannel")
public Object service(Object payload) {
   ...
}
```

Configuring meta-annotations hierarchically lets users set defaults for various attributes and enables isolation of framework Java dependencies to user annotations, avoiding their use in user classes. If the framework finds a method with a user annotation that has a framework meta-annotation, it is treated as if the method were annotated directly with the framework annotation.

F.5.1. Annotations on `@Bean` Methods

Starting with version 4.0, you can configure messaging annotations on `@Bean` method definitions in `@Configuration` classes, to produce message endpoints based on the beans, not the methods. It is useful when `@Bean` definitions are "out-of-the-box" `MessageHandler` instances (`AggregatingMessageHandler`, `DefaultMessageSplitter`, and others), `Transformer` instances (`JsonToObjectTransformer`, `ClaimCheckOutTransformer`, and others), and `MessageSource` instances (`FileReadingMessageSource`, `RedisStoreMessageSource`, and others). The following example shows how to use messaging annotations with `@Bean` annotations:

```
@Configuration
@EnableIntegration
public class MyFlowConfiguration {

    @Bean
    @InboundChannelAdapter(value = "inputChannel", poller = @Poller(fixedDelay =
"1000"))
    public MessageSource<String> consoleSource() {
        return CharacterStreamReadingMessageSource.stdin();
    }

    @Bean
    @Transformer(inputChannel = "inputChannel", outputChannel = "httpChannel")
    public ObjectToMapTransformer toMapTransformer() {
        return new ObjectToMapTransformer();
    }

    @Bean
    @ServiceActivator(inputChannel = "httpChannel")
    public MessageHandler httpHandler() {
    HttpRequestExecutingMessageHandler handler = new
HttpRequestExecutingMessageHandler("https://foo/service");
        handler.setExpectedResponseType(String.class);
        handler.setOutputChannelName("outputChannel");
        return handler;
    }

    @Bean
    @ServiceActivator(inputChannel = "outputChannel")
    public LoggingHandler loggingHandler() {
        return new LoggingHandler("info");
    }

}
```

Version 5.0 introduced support for a `@Bean` annotated with `@InboundChannelAdapter` that returns `java.util.function.Supplier`, which can produce either a POJO or a `Message`. The followig example shows how to use that combination:

```
@Configuration
@EnableIntegration
public class MyFlowConfiguration {

    @Bean
    @InboundChannelAdapter(value = "inputChannel", poller = @Poller(fixedDelay =
"1000"))
    public Supplier<String> pojoSupplier() {
        return () -> "foo";
    }

    @Bean
    @InboundChannelAdapter(value = "inputChannel", poller = @Poller(fixedDelay =
"1000"))
    public Supplier<Message<String>> messageSupplier() {
        return () -> new GenericMessage<>("foo");
    }
}
```

The meta-annotation rules work on `@Bean` methods as well (the `@MyServiceActivator` annotation described earlier can be applied to a `@Bean` definition).

When you use these annotations on consumer `@Bean` definitions, if the bean definition returns an appropriate `MessageHandler` (depending on the annotation type), you must set attributes (such as `outputChannel`, `requiresReply`, `order`, and others), on the `MessageHandler` `@Bean` definition itself. Only the following annotation attributes are used: `adviceChain`, `autoStartup`, `inputChannel`, `phase`, and `poller`. All other attributes are for the handler.

The bean names are generated with the following algorithm:

- The `MessageHandler` (`MessageSource`) `@Bean` gets its own standard name from the method name or `name` attribute on the `@Bean`. This works as though there were no messaging annotation on the `@Bean` method.
- The `AbstractEndpoint` bean name is generated with the following pattern: `[configurationComponentName].[methodName].[decapitalizedAnnotationClassShortName]`. For example, the `SourcePollingChannelAdapter` endpoint for the `consoleSource()` definition shown earlier gets a bean name of `myFlowConfiguration.consoleSource.inboundChannelAdapter`. See also Endpoint Bean Names.

When using these annotations on `@Bean` definitions, the `inputChannel` must reference a declared bean. Channels are not automatically declared in this case.

With Java configuration, you can use any `@Conditional` (for example, `@Profile`) definition on the `@Bean` method level to skip the bean registration for some conditional reason. The following example shows how to do so:

```
@Bean
@ServiceActivator(inputChannel = "skippedChannel")
@Profile("thing")
public MessageHandler skipped() {
    return System.out::println;
}
```

Together with the existing Spring container logic, the messaging endpoint bean (based on the `@ServiceActivator` annotation), is also not registered.

F.5.2. Creating a Bridge with Annotations

Starting with version 4.0, Java configuration provides the `@BridgeFrom` and `@BridgeTo` `@Bean` method annotations to mark `MessageChannel` beans in `@Configuration` classes. These really exists for completeness, providing a convenient mechanism to declare a `BridgeHandler` and its message endpoint configuration:

```
@Bean
public PollableChannel bridgeFromInput() {
    return new QueueChannel();
}

@Bean
@BridgeFrom(value = "bridgeFromInput", poller = @Poller(fixedDelay = "1000"))
public MessageChannel bridgeFromOutput() {
    return new DirectChannel();
}
@Bean
public QueueChannel bridgeToOutput() {
    return new QueueChannel();
}

@Bean
@BridgeTo("bridgeToOutput")
public MessageChannel bridgeToInput() {
    return new DirectChannel();
}
```

You can use these annotations as meta-annotations as well.

F.5.3. Advising Annotated Endpoints

See Advising Endpoints Using Annotations.

F.6. Message Mapping Rules and Conventions

Spring Integration implements a flexible facility to map messages to methods and their arguments without providing extra configuration, by relying on some default rules and defining certain conventions. The examples in the following sections articulate the rules.

F.6.1. Sample Scenarios

The following example shows a single un-annotated parameter (object or primitive) that is not a `Map` or a `Properties` object with a non-void return type:

```
public String doSomething(Object o);
```

The input parameter is a message payload. If the parameter type is not compatible with a message payload, an attempt is made to convert it by using a conversion service provided by Spring 3.0. The return value is incorporated as a payload of the returned message.

The following example shows a single un-annotated parameter (object or primitive)that is not a `Map` or a `Properties` with a `Message` return type:

```
public Message doSomething(Object o);
```

The input parameter is a message payload. If the parameter type is not compatible with a message payload, an attempt is made to convert it by using a conversion service provided by Spring 3.0. The return value is a newly constructed message that is sent to the next destination.

The followig example shows a single parameter that is a message (or one of its subclasses) with an arbitrary object or primitive return type:

```
public int doSomething(Message  msg);
```

The input parameter is itself a `Message`. The return value becomes a payload of the `Message` that is sent to the next destination.

The following example shows a single parameter that is a `Message` (or one of its subclasses) with a `Message` (or one of its subclasses) as the return type:

```
public Message doSomething(Message msg);
```

The input parameter is itself a `Message`. The return value is a newly constructed `Message` that is sent to the next destination.

The following example shows a single parameter of type `Map` or `Properties` with a `Message` as the return type:

```
public Message doSomething(Map m);
```

This one is a bit interesting. Although, at first, it might seem like an easy mapping straight to message headers, preference is always given to a `Message` payload. This means that if a `Message` payload is of type `Map`, this input argument represents a `Message` payload. However, if the `Message` payload is not of type `Map`, the conversion service does not try to convert the payload, and the input argument is mapped to message headers.

The following example shows two parameters, where one of them is an arbitrary type (an object or a primitive) that is not a `Map` or a `Properties` object and the other is of type `Map` or `Properties` type (regardless of the return):

```
public Message doSomething(Map h, <T> t);
```

This combination contains two input parameters where one of them is of type `Map`. The non-`Map` parameters (regardless of the order) are mapped to a `Message` payload and the `Map` or `Properties` (regardless of the order) is mapped to message headers, giving you a nice POJO way of interacting with `Message` structure.

The following example shows no parameters (regardless of the return):

```
public String doSomething();
```

This message handler method is invoked based on the Message sent to the input channel to which this handler is connected. However no `Message` data is mapped, thus making the `Message` act as event or trigger to invoke the handler. The output is mapped according to the rules described earlier.

The following example shows no parameters and a void return:

```
public void soSomething();
```

This example is the same as the previous example, but it produces no output.

F.6.2. Annotation-based Mapping

Annotation-based mapping is the safest and least ambiguous approach to map messages to methods. The following example shows how to explicitly map a method to a header:

```
public String doSomething(@Payload String s, @Header("someheader") String b)
```

As you can see later on, without an annotation this signature would result in an ambiguous condition. However, by explicitly mapping the first argument to a `Message` payload and the second argument to a value of the `someheader` message header, we avoid any ambiguity.

The following example is nearly identical to the preceding example:

```
public String doSomething(@Payload String s, @RequestParam("something") String b)
```

`@RequestMapping` or any other non-Spring Integration mapping annotation is irrelevant and is therefore ignored, leaving the second parameter unmapped. Although the second parameter could easily be mapped to a payload, there can only be one payload. Therefore, the annotations keep this method from being ambiguous.

The following example shows another similar method that would be ambiguous were it not for annotations to clarify the intent:

```
public String foo(String s, @Header("foo") String b)
```

The only difference is that the first argument is implicitly mapped to the message payload.

The following example shows yet another signature that would definitely be treated as ambiguous without annotations, because it has more than two arguments:

```
public String soSomething(@Headers Map m, @Header("something") Map f, @Header(
"someotherthing") String bar)
```

This example would be especially problematic, because two of its arguments are `Map` instances.

However, with annotation-based mapping, the ambiguity is easily avoided. In this example the first argument is mapped to all the message headers, while the second and third argument map to the values of the message headers named 'something' and 'someotherthing'. The payload is not being mapped to any argument.

F.6.3. Complex Scenarios

The following example uses multiple parameters:

Multiple parameters can create a lot of ambiguity with regards to determining the appropriate mappings. The general advice is to annotate your method parameters with `@Payload`, `@Header`, and `@Headers`. The examples in this section show ambiguous conditions that result in an exception being raised.

```
public String doSomething(String s, int i)
```

The two parameters are equal in weight. Therefore, there is no way to determine which one is a payload.

The following example shows a similar problem, only with three parameters:

```
public String foo(String s, Map m, String b)
```

Although the Map could be easily mapped to message headers, there is no way to determine what to do with the two String parameters.

The following example shows another ambiguous method:

```
public String foo(Map m, Map f)
```

Although one might argue that one `Map` could be mapped to the message payload and the other one to the message headers, we cannot rely on the order.

Any method signature with more than one method argument that is not (Map, <T>) and with unannotated parameters results in an ambiguous condition and triggers an exception.

The next set of examples each show mutliple methods that result in ambiguity.

Message handlers with multiple methods are mapped based on the same rules that are described earlier (in the examples). However, some scenarios might still look confusing.

The following example shows multiple methods with legal (mappable and unambiguous) signatures:

```
public class Something {
    public String doSomething(String str, Map m);

    public String doSomething(Map m);
}
```

(Whether the methods have the same name or different names makes no difference). The `Message` could be mapped to either method. The first method would be invoked when the message payload could be mapped to `str` and the message headers could be mapped to `m`. The second method could also be a candidate by mapping only the message headers to `m`. To make matters worse, both methods have the same name. At first, that might look ambiguous because of the following configuration:

```
<int:service-activator input-channel="input" output-channel="output" method=
"doSomething">
    <bean class="org.things.Something"/>
</int:service-activator>
```

It works because mappings are based on the payload first and everything else next. In other words, the method whose first argument can be mapped to a payload takes precedence over all other methods.

Now consider an alternate example, which produces a truly ambiguous condition:

```
public class Something {
    public String doSomething(String str, Map m);

    public String doSomething(String str);
}
```

Both methods have signatures that could be mapped to a message payload. They also have the same name. Such handler methods will trigger an exception. However, if the method names were different, you could influence the mapping with a `method` attribute (shown in the next example). The following example shows the same example with two different method names:

```
public class Something {
    public String doSomething(String str, Map m);

    public String doSomethingElse(String str);
}
```

The following example shows how to use the `method` attribute to dictate the mapping:

```
<int:service-activator input-channel="input" output-channel="output" method=
"doSomethingElse">
    <bean class="org.bar.Foo"/>
</int:service-activator>
```

Because the configuration explicitly maps the `doSomethingElse` method, we have eliminated the ambiguity.

Appendix G: Testing support

Spring Integration provides a number of utilities and annotations to help you test your application. Testing support is presented by two modules:

- `spring-integration-test-support` contains core items and shared utilities
- `spring-integration-test` provides mocking and application context configuration components for integration tests

`spring-integration-test-support` (`spring-integration-test` in versions before 5.0) provides basic, standalone utilities, rules, and matchers for unit testing. (it also has no dependencies on Spring Integration itself and is used internally in Framework tests). `spring-integration-test` aims to help with integration testing and provides a comprehensive high-level API to mock integration components and verify the behavior of individual components, including whole integration flows or only parts of them.

A thorough treatment of testing in the enterprise is beyond the scope of this reference manual. See the "Test-Driven Development in Enterprise Integration Projects" paper, by Gregor Hohpe and Wendy Istvanick, for a source of ideas and principles for testing your target integration solution.

The Spring Integration Test Framework and test utilities are fully based on existing JUnit, Hamcrest, and Mockito libraries. The application context interaction is based on the Spring test framework. See the documentation for those projects for further information.

Thanks to the canonical implementation of the EIP in Spring Integration Framework and its first-class citizens (such as `MessageChannel`, `Endpoint` and `MessageHandler`), abstractions, and loose coupling principles, you can implement integration solutions of any complexity. With the Spring Integration API for the flow definitions, you can improve, modify or even replace some part of the flow without impacting (mostly) other components in the integration solution. Testing such an integration solution is still a challenge, both from an end-to-end approach and from an in-isolation approach. Several existing tools can help to test or mock some integration protocols, and they work well with Spring Integration channel adapters. Examples of such tools include the following:

- Spring `MockMVC` and its `MockRestServiceServer` can be used for testing HTTP.
- Some RDBMS vendors provide embedded data bases for JDBC or JPA support.
- ActiveMQ can be embedded for testing JMS or STOMP protocols.
- There are tools for embedded MongoDB and Redis.
- Tomcat and Jetty have embedded libraries to test real HTTP, Web Services, or WebSockets.
- The `FtpServer` and `SshServer` from the Apache Mina project can be used for testing the FTP and SFTP protocols.
- Gemfire and Hazelcast can be run as real-data grid nodes in the tests.
- The Curator Framework provides a `TestingServer` for Zookeeper interaction.
- Apache Kafka provides admin tools to embed a Kafka Broker in the tests.

Most of these tools and libraries are used in Spring Integration tests. Also, from the GitHub

repository (in the test directory of each module), you can discover ideas for how to build your own tests for integration solutions.

The rest of this chapter describes the testing tools and utilities provided by Spring Integration.

G.1. Testing Utilities

The spring-integration-test-support module provides utilities and helpers for unit testing.

G.1.1. TestUtils

The TestUtils class is mostly used for properties assertions in JUnit tests, as the following example shows:

```
@Test
public void loadBalancerRef() {
    MessageChannel channel = channels.get("lbRefChannel");
    LoadBalancingStrategy lbStrategy = TestUtils.getPropertyValue(channel,
                "dispatcher.loadBalancingStrategy", LoadBalancingStrategy.class);
    assertTrue(lbStrategy instanceof SampleLoadBalancingStrategy);
}
```

TestUtils.getPropertyValue() is based on Spring's DirectFieldAccessor and provides the ability to get a value from the target private property. As shown in the preceding example, it also supports nested properties access by using dotted notation.

The createTestApplicationContext() factory method produces a TestApplicationContext instance with the supplied Spring Integration environment.

See the Javadoc of other TestUtils methods for more information about this class.

G.1.2. Using the SocketUtils Class

The SocketUtils class provides several methods that select one or more random ports for exposing server-side components without conflicts, as the following example shows:

```
<bean id="socketUtils" class="org.springframework.util.SocketUtils" />

<int-syslog:inbound-channel-adapter id="syslog"
            channel="sysLogs"
            port="#{socketUtils.findAvailableUdpPort(1514)}" />

<int:channel id="sysLogs">
    <int:queue/>
</int:channel>
```

The following example shows how the preceding configuration is used from the unit test:

```
@Autowired @Qualifier("syslog.adapter")
private UdpSyslogReceivingChannelAdapter adapter;

@Autowired
private PollableChannel sysLogs;
...
@Test
public void testSimplestUdp() throws Exception {
    int port = TestUtils.getPropertyValue(adapter1, "udpAdapter.port", Integer
.class);
    byte[] buf = "<157>JUL 26 22:08:35 WEBERN TESTING[70729]: TEST SYSLOG MESSAGE
".getBytes("UTF-8");
    DatagramPacket packet = new DatagramPacket(buf, buf.length,
                                new InetSocketAddress("localhost", port));
    DatagramSocket socket = new DatagramSocket();
    socket.send(packet);
    socket.close();
    Message<?> message = foo.receive(10000);
    assertNotNull(message);
}
```

This technique is not foolproof. Some other process could be allocated the “free” port before your test opens it. It is generally more preferable to use server port `0`, let the operating system select the port for you, and then discover the selected port in your test. We have converted most framework tests to use this preferred technique.

G.1.3. Using `OnlyOnceTrigger`

`OnlyOnceTrigger` is useful for polling endpoints when you need to produce only one test message and verify the behavior without impacting other period messages. The following example shows how to configure `OnlyOnceTrigger`:

```
<bean id="testTrigger" class=
"org.springframework.integration.test.util.OnlyOnceTrigger" />

<int:poller id="jpaPoller" trigger="testTrigger">
    <int:transactional transaction-manager="transactionManager" />
</int:poller>
```

The following example shows how to use the preceding configuration of `OnlyOnceTrigger` for testing:

```
@Autowired
@Qualifier("jpaPoller")
PollerMetadata poller;

@Autowired
OnlyOnceTrigger testTrigger;
...
@Test
@DirtiesContext
public void testWithEntityClass() throws Exception {
    this.testTrigger.reset();
    ...
    JpaPollingChannelAdapter jpaPollingChannelAdapter = new
JpaPollingChannelAdapter(jpaExecutor);

    SourcePollingChannelAdapter adapter = JpaTestUtils
.getSourcePollingChannelAdapter(
            jpaPollingChannelAdapter, this.outputChannel, this.poller, this
.context,
            this.getClass().getClassLoader());
    adapter.start();
    ...
}
```

G.1.4. Support Components

The `org.springframework.integration.test.support` package contains various abstract classes that you should implement in target tests

- `AbstractRequestResponseScenarioTests`
- `AbstractResponseValidator`
- `LogAdjustingTestSupport` (Deprecated)
- `MessageValidator`
- `PayloadValidator`

- `RequestResponseScenario`
- `SingleRequestResponseScenarioTests`

G.1.5. JUnit Rules and Conditions

The `LongRunningIntegrationTest` JUnit 4 test rule is present to indicate if test should be run if `RUN_LONG_INTEGRATION_TESTS` environment or system property is set to `true`. Otherwise it is skipped. For the same reason since version 5.1, a `@LongRunningTest` conditional annotation is provided for JUnit 5 tests.

G.1.6. Hamcrest and Mockito Matchers

The `org.springframework.integration.test.matcher` package contains several `Matcher` implementations to assert `Message` and its properties in unit tests. The following example shows how to use one such matcher (`PayloadMatcher`):

```
import static
org.springframework.integration.test.matcher.PayloadMatcher.hasPayload;
...
@Test
public void transform_withFilePayload_convertedToByteArray() throws Exception {
    Message<?> result = this.transformer.transform(message);
    assertThat(result, is(notNullValue()));
    assertThat(result, hasPayload(is(instanceOf(byte[].class))));
    assertThat(result, hasPayload(SAMPLE_CONTENT.getBytes(DEFAULT_ENCODING)));
}
```

The `MockitoMessageMatchers` factory can be used for mocks for stubbing and verifications, as the following example shows:

```
static final Date SOME_PAYLOAD = new Date();

static final String SOME_HEADER_VALUE = "bar";

static final String SOME_HEADER_KEY = "test.foo";
...
Message<?> message = MessageBuilder.withPayload(SOME_PAYLOAD)
                .setHeader(SOME_HEADER_KEY, SOME_HEADER_VALUE)
                .build();
MessageHandler handler = mock(MessageHandler.class);
handler.handleMessage(message);
verify(handler).handleMessage(messageWithPayload(SOME_PAYLOAD));
verify(handler).handleMessage(messageWithPayload(is(instanceOf(Date.class))));
...
MessageChannel channel = mock(MessageChannel.class);
when(channel.send(messageWithHeaderEntry(SOME_HEADER_KEY, is(instanceOf(Short
.class)))))
        .thenReturn(true);
assertThat(channel.send(message), is(false));
```

G.1.7. AssertJ conditions and predicates

Starting with version 5.2, the `MessagePredicate` is introduced to be used in the AssertJ `matches()` assertion. It requires a `Message` object as an expectation. And also ot can be configured with headers to exclude from expectation as well as from actual message to assert.

G.2. Spring Integration and the Test Context

Typically, tests for Spring applications use the Spring Test Framework. Since Spring Integration is based on the Spring Framework foundation, everything we can do with the Spring Test Framework also applies when testing integration flows. The `org.springframework.integration.test.context` package provides some components for enhancing the test context for integration needs. First of all, we configure our test class with a `@SpringIntegrationTest` annotation to enable the Spring Integration Test Framework, as the following example shows:

```
@RunWith(SpringRunner.class)
@SpringIntegrationTest(noAutoStartup = {"inboundChannelAdapter", "*Source*"})
public class MyIntegrationTests {

    @Autowired
    private MockIntegrationContext mockIntegrationContext;

}
```

The `@SpringIntegrationTest` annotation populates a `MockIntegrationContext` bean, which you can autowire to the test class to access its methods. With the `noAutoStartup` option, the Spring Integration Test Framework prevents endpoints that are normally `autoStartup=true` from starting. The endpoints are matched to the provided patterns, which support the following simple pattern styles: `xxx*`, `xxx`, `*xxx`, and `xxx*yyy`.

This is useful when we would like to not have real connections to the target systems from inbound channel adapters (for example an AMQP Inbound Gateway, JDBC Polling Channel Adapter, WebSocket Message Producer in client mode, and so on).

The `MockIntegrationContext` is meant to be used in the target test cases for modifications to beans in the real application context. For example, endpoints that have `autoStartup` overridden to `false` can be replaced with mocks, as the following example shows:

```
@Test
public void testMockMessageSource() {
    MessageSource<String> messageSource = () -> new GenericMessage<>("foo");

    this.mockIntegrationContext.substituteMessageSourceFor("mySourceEndpoint",
messageSource);

    Message<?> receive = this.results.receive(10_000);
    assertNotNull(receive);
}
```

The `mySourceEndpoint` refers here to the bean name of the `SourcePollingChannelAdapter` for which we replace the real `MessageSource` with our mock. Similarly the `MockIntegrationContext.substituteMessageHandlerFor()` expects a bean name for the `IntegrationConsumer`, which wraps a `MessageHandler` as an endpoint.

After test is performed you can restore the state of endpoint beans to the real configuration using `MockIntegrationContext.resetBeans()`:

```
@After
public void tearDown() {
    this.mockIntegrationContext.resetBeans();
}
```

See the Javadoc for more information.

G.3. Integration Mocks

The `org.springframework.integration.test.mock` package offers tools and utilities for mocking, stubbing, and verification of activity on Spring Integration components. The mocking functionality is fully based on and compatible with the well known Mockito Framework. (The current Mockito transitive dependency is on version 2.5.x or higher.)

G.3.1. MockIntegration

The `MockIntegration` factory provides an API to build mocks for Spring Integration beans that are parts of the integration flow (`MessageSource`, `MessageProducer`, `MessageHandler`, and `MessageChannel`). You can use the target mocks during the configuration phase as well as in the target test method to replace the real endpoints before performing verifications and assertions, as the following example shows:

```
<int:inbound-channel-adapter id="inboundChannelAdapter" channel="results">
    <bean class="org.springframework.integration.test.mock.MockIntegration"
factory-method="mockMessageSource">
        <constructor-arg value="a"/>
        <constructor-arg>
            <array>
                <value>b</value>
                <value>c</value>
            </array>
        </constructor-arg>
    </bean>
</int:inbound-channel-adapter>
```

The following example shows how to use Java Configuration to achieve the same configuration as the preceding example:

```
@InboundChannelAdapter(channel = "results")
@Bean
public MessageSource<Integer> testingMessageSource() {
    return MockIntegration.mockMessageSource(1, 2, 3);
}
...
StandardIntegrationFlow flow = IntegrationFlows
        .from(MockIntegration.mockMessageSource("foo", "bar", "baz"))
        .<String, String>transform(String::toUpperCase)
        .channel(out)
        .get();
IntegrationFlowRegistration registration = this.integrationFlowContext
.registration(flow)
        .register();
```

For this purpose, the aforementioned `MockIntegrationContext` should be used from the test, as the following example shows:

```
this.mockIntegrationContext.substituteMessageSourceFor("mySourceEndpoint",
        MockIntegration.mockMessageSource("foo", "bar", "baz"));
Message<?> receive = this.results.receive(10_000);
assertNotNull(receive);
assertEquals("FOO", receive.getPayload());
```

Unlike the Mockito `MessageSource` mock object, the `MockMessageHandler` is a regular `AbstractMessageProducingHandler` extension with a chain API to stub handling for incoming messages. The `MockMessageHandler` provides `handleNext(Consumer<Message<?>>)` to specify a one-way stub for the next request message. It is used to mock message handlers that do not produce replies. `handleNextAndReply(Function<Message<?>, ?>)` is provided for performing the same stub logic for the next request message and producing a reply for it. They can be chained to simulate any arbitrary request-reply scenarios for all expected request messages variants. These consumers and functions are applied to the incoming messages, one at a time from the stack, until the last, which is then used for all remaining messages. The behavior is similar to the Mockito `Answer` or `doReturn()` API.

In addition, you can supply a Mockito `ArgumentCaptor<Message<?>>` to the `MockMessageHandler` in a constructor argument. Each request message for the `MockMessageHandler` is captured by that `ArgumentCaptor`. During the test, you can use its `getValue()` and `getAllValues()` methods to verify and assert those request messages.

The `MockIntegrationContext` provides a `substituteMessageHandlerFor()` API that lets you replace the actual configured `MessageHandler` with a `MockMessageHandler` in the endpoint under test.

The following example shows a typical usage scenario:

```
ArgumentCaptor<Message<?>> messageArgumentCaptor = ArgumentCaptor.forClass(
Message.class);

MessageHandler mockMessageHandler =
        mockMessageHandler(messageArgumentCaptor)
                .handleNextAndReply(m -> m.getPayload().toString().toUpperCase());

this.mockIntegrationContext.substituteMessageHandlerFor("myService.serviceActivato
r",
                                mockMessageHandler);
GenericMessage<String> message = new GenericMessage<>("foo");
this.myChannel.send(message);
Message<?> received = this.results.receive(10000);
assertNotNull(received);
assertEquals("FOO", received.getPayload());
assertSame(message, messageArgumentCaptor.getValue());
```

See the `MockIntegration` and `MockMessageHandler` Javadoc for more information.

G.4. Other Resources

As well as exploring the test cases in the framework itself, the Spring Integration Samples repository has some sample applications specifically made to show testing, such as `testing-examples` and `advanced-testing-examples`. In some cases, the samples themselves have comprehensive end-to-end tests, such as the `file-split-ftp` sample.

Appendix H: Spring Integration Samples

As of Spring Integration 2.0, the Spring Integration distribution no longer includes the samples. Instead, we have switched to a much simpler collaborative model that should promote better community participation and, ideally, more contributions. Samples now have a dedicated Git repository and a dedicated JIRA Issue Tracking system. Sample development also has its own lifecycle, which is not dependent on the lifecycle of the framework releases, although the repository is still tagged with each major release for compatibility reasons.

The great benefit to the community is that we can now add more samples and make them available to you right away without waiting for the next release. Having its own JIRA that is not tied to the the actual framework is also a great benefit. You now have a dedicated place to suggest samples as well as report issues with existing samples. You can also submit a sample to us as an attachment through JIRA or, better, through the collaborative model that Git promotes. If we believe your sample adds value, we would be more then glad to add it to the 'samples' repository, properly crediting you as the author.

H.1. Where to Get Samples

The Spring Integration Samples project is hosted on GitHub. You can find the repository at:

https://github.com/SpringSource/spring-integration-samples

In order to check out or clone the samples, you must have a Git client installed on your system. There are several GUI-based products available for many platforms (such as EGit for the Eclipse IDE). A simple Google search can help you find them. You can also use the command line interface for Git.

If you need more information on how to install or use Git, visit: https://git-scm.com/.

To clone (check out) the Spring Integration samples repository by using the Git command line tool, issue the following command:

```
$ git clone https://github.com/SpringSource/spring-integration-samples.git
```

The preceding command clones the entire samples repository into a directory named `spring-integration-samples` within the working directory where you issued that `git` command. Since the samples repository is a live repository, you might want to perform periodic pulls (updates) to get new samples and updates to the existing samples. To do so, issue the following `git pull` command:

```
$ git pull
```

H.2. Submitting Samples or Sample Requests

You can submit both new samples and requests for samples. We greatly appreciate any effort toward improving the samples, including the sharing of good ideas.

H.2.1. How Can I Contribute My Own Samples?

Github is for social coding: if you want to submit your own code examples to the Spring Integration Samples project, we encourage contributions through pull requests from forks of this repository. If you want to contribute code this way, please reference, if possible, a JIRA ticket that provides some details regarding your sample.

Sign the contributor license agreement

Very important: Before we can accept your Spring Integration sample, we need you to sign the SpringSource contributor license agreement (CLA). Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. In order to read and sign the CLA, go to:

support.springsource.com/spring_committer_signup

From the **Project** drop down, select **Spring Integration**. The Project Lead is Gary Russell.

H.2.2. Code Contribution Process

For the actual code contribution process, read the the Contributor Guidelines for Spring Integration. They apply for the samples project as well. You can find them at github.com/spring-projects/spring-integration/blob/master/CONTRIBUTING.md

This process ensures that every commit gets peer-reviewed. As a matter of fact, the core committers follow the exact same rules. We gratefully look forward to your Spring Integration samples!

H.2.3. Sample Requests

As mentioned earlier, the Spring Integration Samples project has a dedicated JIRA issue tracking system. To submit new sample requests, visit the JIRA Issue Tracking system at jira.springframework.org/browse/INTSAMPLES.

H.3. Samples Structure

Starting with Spring Integration 2.0, the structure of the samples has changed. With plans for more samples, we realized that not all samples have the same goals. They all share the common goal of showing you how to apply and work with the Spring Integration framework. However, they differ in that some samples concentrate on a technical use case, while others focus on a business use case. Also, some samples are about showcasing various techniques that could be applied to address certain scenarios (both technical and business). The new categorization of samples lets us better organize them based on the problem each sample addresses while giving you a simpler way of

finding the right sample for your needs.

Currently there are four categories. Within the samples repository, each category has its own directory, which is named after the category name:

Basic (`samples/basic`)

This is a good place to get started. The samples here are technically motivated and demonstrate the bare minimum with regard to configuration and code. These should help you to get started quickly by introducing you to the basic concepts, API, and configuration of Spring Integration as well as Enterprise Integration Patterns (EIP). For example, if you are looking for an answer on how to implement and wire a service activator to a message channel, how to use a messaging gateway as a facade to your message exchange, or how to get started with MAIL, TCP/UDP or other modules, this is the right place to find a good sample. The bottom line is `samples/basic` is a good place to get started.

Intermediate (`samples/intermediate`)

This category targets developers who are already familiar with the Spring Integration framework (beyond getting started) but need some more guidance while resolving the more advanced technical problems they might encounter after switching to a messaging architecture. For example, if you are looking for an answer on how to handle errors in various message exchange scenarios or how to properly configure the aggregator for a situation where some messages do not ever arrive for aggregation, or any other issue that goes beyond a basic implementation and configuration of a particular component and exposes "what else" types of problems, this is the right place to find these type of samples.

Advanced (`samples/advanced`)

This category targets developers who are very familiar with the Spring Integration framework but are looking to extend it to address a specific custom need by using Spring Integration's public API. For example, if you are looking for samples showing you how to implement a custom channel or consumer (event-based or polling-based) or you are trying to figure out the most appropriate way to implement a custom bean parser on top of the Spring Integration bean parser hierarchy (perhaps when implementing your own namespace and schema for a custom component), this is the right place to look. Here you can also find samples that will help you with adapter development. Spring Integration comes with an extensive library of adapters to let you connect remote systems with the Spring Integration messaging framework. However, you might need to integrate with a system for which the core framework does not provide an adapter. If so, you might decide to implement your own (please consider contributing it). This category would include samples showing you how.

Applications (`samples/applications`)

This category targets developers and architects who have a good understanding of message-driven architecture and EIP and an above-average understanding of Spring and Spring Integration who are looking for samples that address a particular business problem. In other words, the emphasis of the samples in this category is business use cases and how they can be solved with a message-driven architecture and Spring Integration in particular. For example, if you want to see how a loan broker or travel agent process could be implemented and automated with Spring Integration, this is the right place to find these types of samples.

Spring Integration is a community-driven framework. Therefore community participation is IMPORTANT. That includes samples. If you cannot find what you are looking for, let us know!

H.4. Samples

Currently, Spring Integration comes with quite a few samples and you can only expect more. To help you better navigate through them, each sample comes with its own `readme.txt` file which covers several details about the sample (for example, what EIP patterns it addresses, what problem it is trying to solve, how to run the sample, and other details). However, certain samples require a more detailed and sometimes graphical explanation. In this section, you can find details on samples that we believe require special attention.

H.4.1. Loan Broker

This section covers the loan broker sample application that is included in the Spring Integration samples. This sample is inspired by one of the samples featured in Gregor Hohpe and Bobby Woolf's book, *Enterprise Integration Patterns*.

The following diagram shows the entire process:

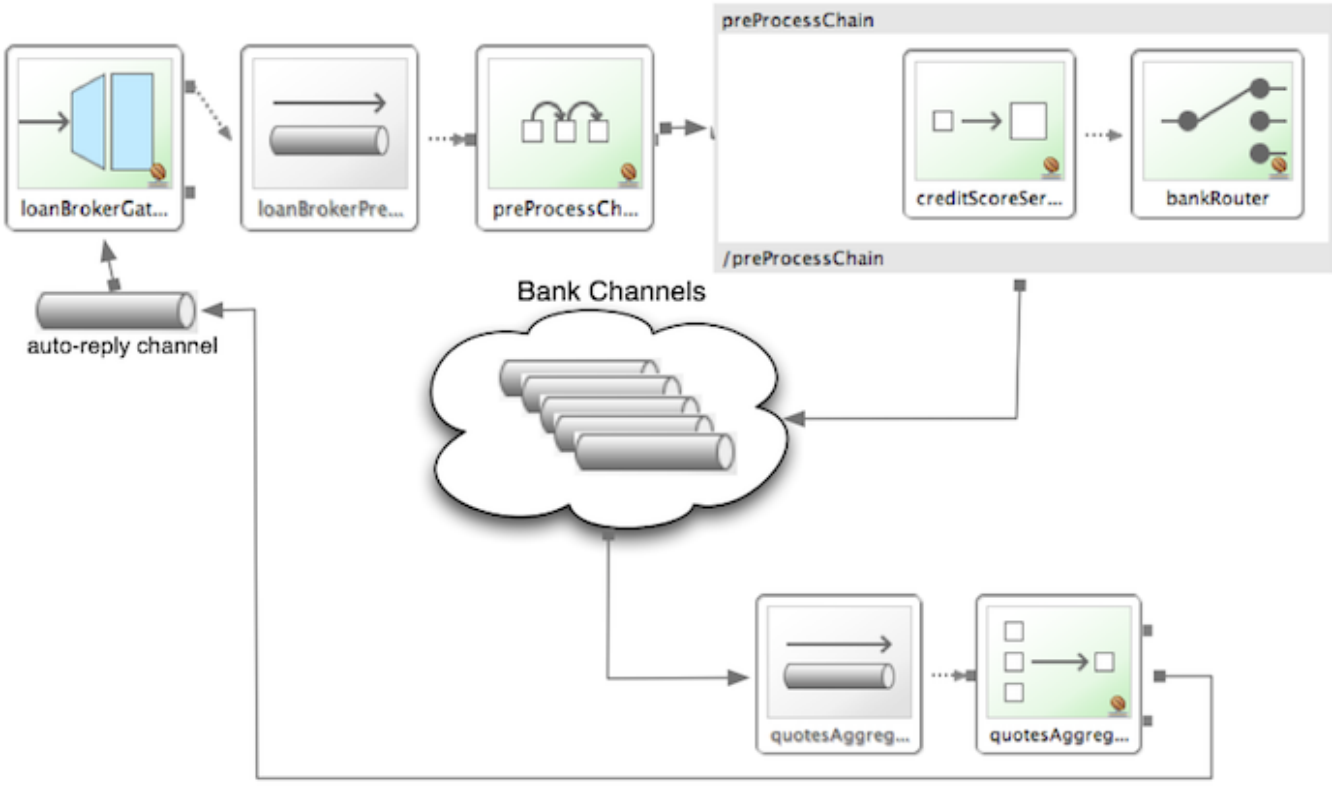


Figure 10. Loan Broker Sample

At the core of an EIP architecture are the very simple yet powerful concepts of pipes, filters, and, of course: messages. Endpoints (filters) are connected with one another via channels (pipes). Producing endpoints send messages to the channel, and the consuming endpoint retrieves the messages. This architecture is meant to define various mechanisms that describe how information is exchanged between the endpoints, without any awareness of what those endpoints are or what

information they are exchanging. Thus, it provides for a very loosely coupled and flexible collaboration model while also decoupling integration concerns from business concerns. EIP extends this architecture by further defining:

- The types of pipes (point-to-point channel, publish-subscribe channel, channel adapter, and others)
- The core filters and patterns around how filters collaborate with pipes (Message router, splitters and aggregators, various message transformation patterns, and others)

Chapter 9 of the EIP book nicely describes the details and variations of this use case, but here is the brief summary: While shopping for the best loan quote, a consumer subscribes to the services of a loan broker, which handles such details as:

- Consumer pre-screening (for example, obtaining and reviewing the consumer's Credit history)
- Determining the most appropriate banks (for example, based on the consumer's credit history or score)
- Sending a loan quote request to each selected bank
- Collecting responses from each bank
- Filtering responses and determining the best quotes, based on consumer's requirements.
- Pass the Loan quotes back to the consumer.

The real process of obtaining a loan quote is generally a bit more complex. However, since our goal is to demonstrate how Enterprise Integration Patterns are realized and implemented within Spring Integration, the use case has been simplified to concentrate only on the integration aspects of the process. It is not an attempt to give you advice in consumer finances.

By engaging a loan broker, the consumer is isolated from the details of the loan broker's operations, and each loan broker's operations may defer from one another to maintain competitive advantage, so whatever we assemble and implement must be flexible so that any changes could be introduced quickly and painlessly.

The loan broker sample does not actually talk to any 'imaginary' Banks or Credit bureaus. Those services are stubbed out.

Our goal here is to assemble, orchestrate, and test the integration aspects of the process as a whole. Only then can we start thinking about wiring such processes to the real services. At that time, the assembled process and its configuration do not change regardless of the number of banks with which a particular loan broker deals or the type of communication media (or protocols) used (JMS, WS, TCP, and so on) to communicate with these banks.

Design

As you analyze the six requirements listed earlier, you can see that they are all integration concerns. For example, in the consumer pre-screening step, we need to gather additional information about the consumer and the consumer's desires and enrich the loan request with additional meta-information. We then have to filter such information to select the most appropriate list of banks and so on. Enrich, filter, and select are all integration concerns for which EIP defines a

solution in the form of patterns. Spring Integration provides an implementation of these patterns.

The following image shows a representation of a messaging gateway:

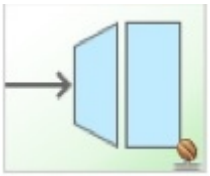


Figure 11. Messaging Gateway

The messaging gateway pattern provides a simple mechanism to access messaging systems, including our loan broker. In Spring Integration, you can define the gateway as a plain old java interface (you need not provide an implementation), configure it with the XML `<gateway>` element or with an annotation in Java, and use it as you would any other Spring bean. Spring Integration takes care of delegating and mapping method invocations to the messaging infrastructure by generating a message (the payload is mapped to an input parameter of the method) and sending it to the designated channel. The following example shows how to define such a gateway with XML:

```
<int:gateway id="loanBrokerGateway"
  default-request-channel="loanBrokerPreProcessingChannel"
  service-interface=
"org.springframework.integration.samples.loanbroker.LoanBrokerGateway">
  <int:method name="getBestLoanQuote">
    <int:header name="RESPONSE_TYPE" value="BEST"/>
  </int:method>
</int:gateway>
```

Our current gateway provides two methods that could be invoked. One that returns the best single quote and another one that returns all quotes. Somehow, downstream, we need to know what type of reply the caller needs. The best way to achieve this in messaging architecture is to enrich the content of the message with some metadata that describes your intentions. Content Enricher is one of the patterns that addresses this. Spring Integration does, as a convenience, provide a separate configuration element to enrich message headers with arbitrary data (described later) However, since the `gateway` element is responsible for constructing the initial message, it includes ability to enrich the newly created message with arbitrary message headers. In our example, we add a `RESPONSE_TYPE` header with a value of `BEST` whenever the `getBestQuote()` method is invoked. For other methods, we do not add any header. Now we can check downstream for the existence of this header. Based on its presence and its value, we can determine what type of reply the caller wants.

Based on the use case, we also know tat some pre-screening steps need to be performed, such as getting and evaluating the consumer's credit score, because some premiere banks only accept quote requests from consumers that meet a minimum credit score requirement. So it would be nice if the message would be enriched with such information before it is forwarded to the banks. It would also be nice if, when several processes need to be completed to provide such meta-information, those processes could be grouped in a single unit. In our use case, we need to determine the credit score and, based on the credit score and some rule, select a list of message

channels (bank channels) to which to send quote request.

Composed Message Processor

The composed message processor pattern describes rules around building endpoints that maintain control over message flow, which consists of multiple message processors. In Spring Integration, the composed message processor pattern is implemented by the `<chain>` element.

The following image shows the chain pattern:

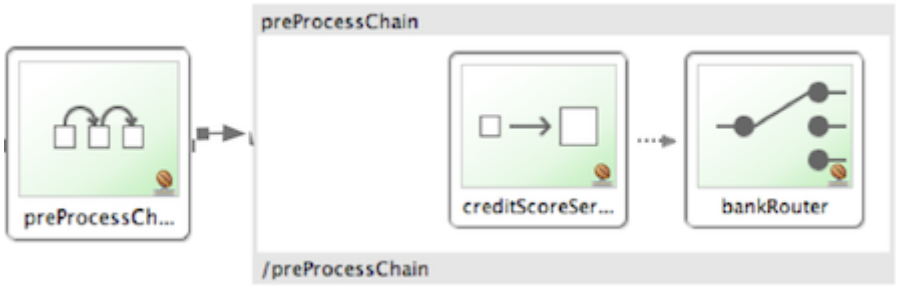


Figure 12. Chain

The preceding image shows that we have a chain with an inner header-enricher element that further enriches the content of the message with the `CREDIT_SCORE` header and the value (which is determined by the call to a credit service — a simple POJO spring bean identified by 'creditBureau' name). Then it delegates to the message router.

The following image shows the message router pattern:

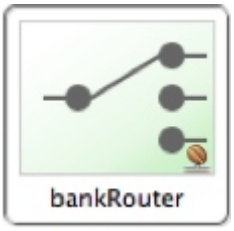


Figure 13. Message Router

Spring Integration offers several implementations of the message routing pattern. In this case, we use a router that determines a list of channels based on evaluating an expression (in Spring Expression Language) that looks at the credit score (determined in the previous step) and selects the list of channels from the `Map` bean with an `id` of `banks` whose values are `premier` or `secondary`, based on the value of credit score. Once the list of channels is selected, the message is routed to those channels.

Now, one last thing the loan broker needs to receive the loan quotes form the banks, aggregate them by consumer (we do not want to show quotes from one consumer to another), assemble the response based on the consumer's selection criteria (single best quote or all quotes) and send the reply to the consumer.

The following image shows the message aggregator pattern:

Figure 14. Message Aggregator

An aggregator pattern describes an endpoint that groups related messages into a single message. Criteria and rules can be provided to determine an aggregation and correlation strategy. Spring Integration provides several implementations of the aggregator pattern as well as a convenient namespace-based configuration.

The following example shows how to define an aggregator:

```
<int:aggregator id="quotesAggregator"
      input-channel="quotesAggregationChannel"
      method="aggregateQuotes">
  <beans:bean class=
"org.springframework.integration.samples.loanbroker.LoanQuoteAggregator"/>
</int:aggregator>
```

Our Loan Broker defines a 'quotesAggregator' bean with the `<aggregator>` element, which provides a default aggregation and correlation strategy. The default correlation strategy correlates messages based on the `correlationId` header (see the correlation identifier pattern in the EIP book). Note that we never provided the value for this header. It was automatically set earlier by the router, when it generated a separate message for each bank channel.

Once the messages are correlated, they are released to the actual aggregator implementation. Although Spring Integration provides a default aggregator, its strategy (gather the list of payloads from all messages and construct a new message with this list as its payload) does not satisfy our requirement. Having all the results in the message is a problem, because our consumer might require a single best quote or all quotes. To communicate the consumer's intention, earlier in the process we set the `RESPONSE_TYPE` header. Now we have to evaluate this header and return either all the quotes (the default aggregation strategy would work) or the best quote (the default aggregation strategy does not work because we have to determine which loan quote is the best).

In a more realistic application, selecting the best quote might be based on complex criteria that might influence the complexity of the aggregator implementation and configuration. For now, though, we are making it simple. If the consumer wants the best quote, we select a quote with the lowest interest rate. To accomplish that, the `LoanQuoteAggregator` class sorts all the quotes by interest rate and returns the first one. The `LoanQuote` class implements `Comparable` to compare quotes based on the rate attribute. Once the response message is created, it is sent to the default reply channel of the messaging gateway (and, thus, to the consumer) that started the process. Our consumer got the loan quote!

In conclusion, a rather complex process was assembled based on POJO (that is existing or legacy)

logic and a light-weight, embeddable messaging framework (Spring Integration) with a loosely coupled programming model intended to simplify integration of heterogeneous systems without requiring a heavy-weight ESB-like engine or a proprietary development and deployment environment. As a developer, you should not need to port your Swing or console-based application to an ESB-like server or implement proprietary interfaces just because you have an integration concern.

This sample and the other samples in this section are built on top of Enterprise Integration Patterns. You can consider them to be “building blocks” for your solution. They are not intended to be complete solutions. Integration concerns exist in all types of application (whether server-based or not). Our goal is to make is so that integrating applications does not require changes in design, testing, and deployment strategy.

H.4.2. The Cafe Sample

This section covers the cafe sample application that is included in the Spring Integration samples. This sample is inspired by another sample featured in Gregor Hohpe’s Ramblings.

The domain is that of a cafe, and the following diagram depicts the basic flow:

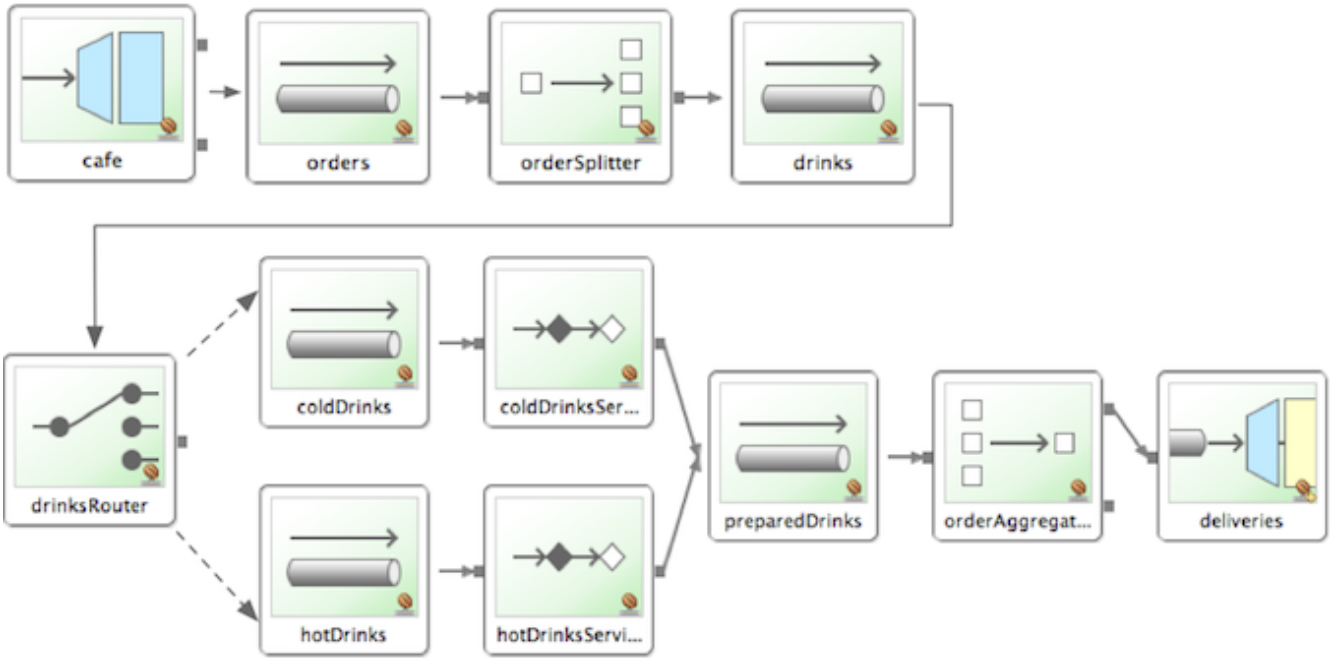


Figure 15. Cafe Sample

The `Order` object may contain multiple `OrderItems`. Once the order is placed, a splitter breaks the composite order message into a single message for each drink. Each of these is then processed by a router that determines whether the drink is hot or cold (by checking the `OrderItem` object’s 'isIced' property). The `Barista` prepares each drink, but hot and cold drink preparation are handled by two distinct methods: 'prepareHotDrink' and 'prepareColdDrink'. The prepared drinks are then sent to the `Waiter` where they are aggregated into a `Delivery` object.

The following listing shows the XML configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:int="http://www.springframework.org/schema/integration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:beans="http://www.springframework.org/schema/beans"
 xmlns:int-stream="http://www.springframework.org/schema/integration/stream"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
  https://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/integration
  https://www.springframework.org/schema/integration/spring-integration.xsd
  http://www.springframework.org/schema/integration/stream
  https://www.springframework.org/schema/integration/stream/spring-integration-
stream.xsd">

    <int:gateway id="cafe" service-interface="o.s.i.samples.cafe.Cafe"/>

    <int:channel  id="orders"/>
    <int:splitter input-channel="orders" ref="orderSplitter"
                  method="split" output-channel="drinks"/>

    <int:channel id="drinks"/>
    <int:router  input-channel="drinks"
                 ref="drinkRouter" method="resolveOrderItemChannel"/>

    <int:channel id="coldDrinks"><int:queue capacity="10"/></int:channel>
    <int:service-activator input-channel="coldDrinks" ref="barista"
                           method="prepareColdDrink" output-channel=
"preparedDrinks"/>

    <int:channel id="hotDrinks"><int:queue capacity="10"/></int:channel>
    <int:service-activator input-channel="hotDrinks" ref="barista"
                           method="prepareHotDrink" output-channel="
preparedDrinks"/>

    <int:channel id="preparedDrinks"/>
    <int:aggregator input-channel="preparedDrinks" ref="waiter"
                    method="prepareDelivery" output-channel="deliveries"/>

    <int-stream:stdout-channel-adapter id="deliveries"/>

    <beans:bean id="orderSplitter"
                class=
"org.springframework.integration.samples.cafe.xml.OrderSplitter"/>

    <beans:bean id="drinkRouter"
                class=
"org.springframework.integration.samples.cafe.xml.DrinkRouter"/>

    <beans:bean id="barista" class="o.s.i.samples.cafe.xml.Barista"/>
    <beans:bean id="waiter"  class="o.s.i.samples.cafe.xml.Waiter"/>
```

```
    <int:poller id="poller" default="true" fixed-rate="1000"/>

</beans:beans>
```

Each message endpoint connects to input channels, output channels, or both. Each endpoint manages its own lifecycle (by default, endpoints start automatically upon initialization, to prevent that, add the `auto-startup` attribute with a value of `false`). Most importantly, notice that the objects are simple POJOs with strongly typed method arguments. The following example shows the Splitter:

```
public class OrderSplitter {
    public List<OrderItem> split(Order order) {
        return order.getItems();
    }
}
```

In the case of the router, the return value does not have to be a `MessageChannel` instance (although it can be). In this example, a `String` value that holds the channel name is returned instead, as the following listing shows.

```
public class DrinkRouter {
    public String resolveOrderItemChannel(OrderItem orderItem) {
        return (orderItem.isIced()) ? "coldDrinks" : "hotDrinks";
    }
}
```

Now, turning back to the XML, you can see that there are two `<service-activator>` elements. Each of these is delegating to the same `Barista` instance but with different methods (`prepareHotDrink` or `prepareColdDrink`), each corresponding to one of the two channels where order items have been routed. The following listing shows the Barista class (which contains the `prepareHotDrink` and `prepareColdDrink` methods)

```
public class Barista {

    private long hotDrinkDelay = 5000;
    private long coldDrinkDelay = 1000;

    private AtomicInteger hotDrinkCounter = new AtomicInteger();
    private AtomicInteger coldDrinkCounter = new AtomicInteger();

    public void setHotDrinkDelay(long hotDrinkDelay) {
        this.hotDrinkDelay = hotDrinkDelay;
    }

    public void setColdDrinkDelay(long coldDrinkDelay) {
        this.coldDrinkDelay = coldDrinkDelay;
    }

    public Drink prepareHotDrink(OrderItem orderItem) {
        try {
            Thread.sleep(this.hotDrinkDelay);
            System.out.println(Thread.currentThread().getName()
                    + " prepared hot drink #" + hotDrinkCounter.incrementAndGet()
                    + " for order #" + orderItem.getOrder().getNumber()
                    + ": " + orderItem);
            return new Drink(orderItem.getOrder().getNumber(), orderItem
.getDrinkType(),
                    orderItem.isIced(), orderItem.getShots());
        }
        catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return null;
        }
    }

    public Drink prepareColdDrink(OrderItem orderItem) {
        try {
            Thread.sleep(this.coldDrinkDelay);
            System.out.println(Thread.currentThread().getName()
                    + " prepared cold drink #" + coldDrinkCounter.incrementAndGet
()
                    + " for order #" + orderItem.getOrder().getNumber() + ": "
                    + orderItem);
            return new Drink(orderItem.getOrder().getNumber(), orderItem
.getDrinkType(),
                    orderItem.isIced(), orderItem.getShots());
        }
        catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return null;
        }
```

```
    }
}
```

As you can see from the preceding code excerpt, the `Barista` methods have different delays (the hot drinks take five times as long to prepare). This simulates work being completed at different rates. When the `CafeDemo` 'main' method runs, it loops 100 times and sends a single hot drink and a single cold drink each time. It actually sends the messages by invoking the 'placeOrder' method on the `Cafe` interface. In the earlier XML configuration, you can see that the `<gateway>` element is specified. This triggers the creation of a proxy that implements the given service interface and connects it to a channel. The channel name is provided on the `@Gateway` annotation of the `Cafe` interface, as the following interface definition shows:

```
public interface Cafe {

    @Gateway(requestChannel="orders")
    void placeOrder(Order order);

}
```

Finally, have a look at the `main()` method of the `CafeDemo` itself:

```
public static void main(String[] args) {
    AbstractApplicationContext context = null;
    if (args.length > 0) {
        context = new FileSystemXmlApplicationContext(args);
    }
    else {
        context = new ClassPathXmlApplicationContext("cafeDemo.xml", CafeDemo
.class);
    }
    Cafe cafe = context.getBean("cafe", Cafe.class);
    for (int i = 1; i <= 100; i++) {
        Order order = new Order(i);
        order.addItem(DrinkType.LATTE, 2, false);
        order.addItem(DrinkType.MOCHA, 3, true);
        cafe.placeOrder(order);
    }
}
```

To run this sample as well as eight others, refer to the `README.txt` within the `samples` directory of the main distribution (as described at the beginning of this chapter).

When you run `cafeDemo`, you can see that the cold drinks are initially prepared more quickly than the hot drinks. Because there is an aggregator, the cold drinks are effectively limited by the rate of the hot drink preparation. This is to be expected, based on their respective delays of 1000 and 5000 milliseconds. However, by configuring a poller with a concurrent task executor, you can dramatically change the results. For example, you could use a thread pool executor with five workers for the hot drink barista while keeping the cold drink barista as it is. The following listing configures such an arrangement:

```
<int:service-activator input-channel="hotDrinks"
                       ref="barista"
                       method="prepareHotDrink"
                       output-channel="preparedDrinks"/>

  <int:service-activator input-channel="hotDrinks"
                       ref="barista"
                       method="prepareHotDrink"
                       output-channel="preparedDrinks">
      <int:poller task-executor="pool" fixed-rate="1000"/>
  </int:service-activator>

  <task:executor id="pool" pool-size="5"/>
```

Also, notice that the worker thread name is displayed with each invocation. You can see that the hot drinks are prepared by the task-executor threads. If you provide a much shorter poller interval (such as 100 milliseconds), you can see that it occasionally throttles the input by forcing the task scheduler (the caller) to invoke the operation.

In addition to experimenting with the poller's concurrency settings, you can also add the 'transactional' child element and then refer to any `PlatformTransactionManager` instance within the context.

H.4.3. The XML Messaging Sample

The XML messaging sample in `basic/xml` shows how to use some of the provided components that deal with XML payloads. The sample uses the idea of processing an order for books represented as XML.

This sample shows that the namespace prefix can be whatever you want. While we usually use, `int-xml` for integration XML components, the sample uses `si-xml`. (`int` is short for "Integration", and `si` is short for "Spring Integration".)

First, the order is split into a number of messages, each one representing a single order item from the XPath splitter component. The following listing shows the configuration of the splitter:

```
<si-xml:xpath-splitter id="orderItemSplitter" input-channel="ordersChannel"
              output-channel="stockCheckerChannel" create-documents="true">
      <si-xml:xpath-expression expression="/orderNs:order/orderNs:orderItem"
                               namespace-map="orderNamespaceMap" />
  </si-xml:xpath-splitter>
```

A service activator then passes the message into a stock checker POJO. The order item document is enriched with information from the stock checker about the order item stock level. This enriched order item message is then used to route the message. In the case where the order item is in stock, the message is routed to the warehouse. The following listing configures the `xpath-router` that routes the messages:

```
<si-xml:xpath-router id="instockRouter" input-channel="orderRoutingChannel"
resolution-required="true">
    <si-xml:xpath-expression expression="/orderNs:orderItem/@in-stock" namespace-
map="orderNamespaceMap" />
    <si-xml:mapping value="true" channel="warehouseDispatchChannel"/>
    <si-xml:mapping value="false" channel="outOfStockChannel"/>
</si-xml:xpath-router>
```

When the order item is not in stock, the message is transformed with XSLT into a format suitable for sending to the supplier. The following listing configures the XSLT transformer:

```
<si-xml:xslt-transformer input-channel="outOfStockChannel"
  output-channel="resupplyOrderChannel"
  xsl-resource=
"classpath:org/springframework/integration/samples/xml/bigBooksSupplierTransformer
.xsl"/>
```

Appendix I: Additional Resources

The definitive source of information about Spring Integration is the Spring Integration Home at https://spring.io. That site serves as a hub of information and is the best place to find up-to-date announcements about the project as well as links to articles, blogs, and new sample applications.

Appendix J: Change History

J.1. Changes between 5.1 and 5.2

J.2. Package and Class Changes

`Pausable` has been moved from `o.s.i.endpoint` to `o.s.i.core`.

J.3. Behavior Changes

See the Migration Guide about behavior changes in this version.

J.4. New Components

J.4.1. RSocket Support

The `spring-integration-rsocket` module is now available with channel adapter implementations for RSocket protocol support. See RSocket Support for more information.

J.4.2. Rate Limit Advice Support

The `RateLimiterRequestHandlerAdvice` is now available for limiting requests rate on handlers. See Rate Limiter Advice for more information.

J.4.3. Caching Advice Support

The `CacheRequestHandlerAdvice` is now available for caching request results on handlers. See Caching Advice for more information.

J.4.4. Kotlin Scripts Support

The JSR223 scripting module now includes a support for Kotlin scripts. See Scripting Support for more information.

J.4.5. Flux Aggregator Support

The `FluxAggregatorMessageHandler` is now available for grouping and windowing messages logic based on the Project Reactor `Flux` operators. See Flux Aggregator for more information.

J.4.6. FTP/SFTP Event Publisher

The FTP and SFTP modules now provide an event listener for certain Apache Mina FTP/SFTP server events. See Apache Mina FTP Server Events and Apache Mina SFTP Server Events for more information.

J.4.7. Avro Transformers

Simple Apache Avro transformers are now provided. See Avro Transformers for more information.

J.5. General Changes

The `JsonToObjectTransformer` now supports generics for the target object to deserialize into. See JSON Transformers for more information.

The `splitter` now supports a `discardChannel` configuration option. See Splitter for more information.

The Control Bus can now handle `Pausable` (extension of `Lifecycle`) operations. See Control Bus for more information.

The `Function<MessageGroup, Map<String, Object>>` strategy has been introduced for the aggregator component to merge and compute headers for output messages. See Aggregator Programming Model for more information.

All the `MessageHandlingException` s thrown in the framework, includes now a bean resource and source for back tracking a configuration part in case no end-user code involved. See Error Handling for more information.

For better end-user experience, Java DSL now provides a configurer variant for starting flow with a gateway interface. See `IntegrationFlows.from(Class<?> serviceInterface, Consumer<GatewayProxySpec> endpointConfigurer)` JavaDocs for more information. Also a `MethodArgsHolder` is now a root object for evaluation context for all the expressions in the `GatewayProxyFactoryBean`. The `#args` and `#method` evaluation context variables are now deprecated. See Messaging Gateways for more information.

J.5.1. AMQP Changes

The outbound endpoints can now be configured to synthesize a "nack" if no publisher confirm is received within a timeout. See Outbound Channel Adapter for more information.

The inbound channel adapter can now receive batched messages as a `List<?>` payload instead of receiving a discrete message for each batch fragment. See Batched Messages for more information.

The outbound channel adapter can now be configured to block the calling thread until a publisher confirm (acknowledgment) is received. See Outbound Channel Adapter for more information.

J.5.2. File Changes

Some improvements to filtering remote files have been made. See Remote Persistent File List Filters for more information.

J.5.3. TCP Changes

The length header used by the `ByteArrayLengthHeaderSerializer` can now include the length of the header in addition to the payload. See Message Demarcation (Serializers and Deserializers) for

more information.

When using a `TcpNioServerConnectionFactory`, priority is now given to accepting new connections over reading from existing connections, but it is configurable. See About Non-blocking I/O (NIO) for more information.

The outbound gateway has a new property `closeStreamAfterSend`; when used with a new connection for each request/reply it signals EOF to the server, without closing the connection. This is useful for servers that use the EOF to signal end of message instead of some delimiter in the data. See TCP Gateways for more information.

The client connection factories now support `connectTimeout` which causes an exception to be thrown if the connection is not established in that time. See TCP Connection Factories for more information.

`SoftEndOfStreamException` is now a `RuntimeException` instead of extending `IOException`.

J.5.4. Mail Changes

The `AbstractMailReceiver` has now an `autoCloseFolder` option (`true` by default), to disable an automatic folder close after a fetch, but populate `IntegrationMessageHeaderAccessor.CLOSEABLE_RESOURCE` header instead for downstream interaction. See Mail-receiving Channel Adapter for more information.

J.5.5. HTTP Changes

The HTTP inbound endpoint now support a request payload validation. See HTTP Support for more information.

J.5.6. WebFlux Changes

The `WebFluxRequestExecutingMessageHandler` now supports a `Publisher`, `Resource` and `MultiValueMap` as a request message `payload`. The `WebFluxInboundEndpoint` now supports a request payload validation. See WebFlux Support for more information.

J.5.7. MongoDb Changes

The `MongoDbMessageStore` can now be configured with custom converters. See MongoDB Support for more information.

J.5.8. Router Changes

You can now disable falling back to the channel key as the channel bean name. See Dynamic Routers for more information.

J.5.9. FTP/SFTP Changes

The `RotatingServerAdvice` is decoupled now from the `RotationPolicy` and its `StandardRotationPolicy`.

The remote file information, including host/port and directory are included now into message

headers in the `AbstractInboundFileSynchronizingMessageSource` and `AbstractRemoteFileStreamingMessageSource` implementations. Also this information is included into headers in the read operations results of the `AbstractRemoteFileOutboundGateway` implementations. The FTP outbound endpoints now support `chmod` to change permissions on the uploaded file. (SFTP already supported it since version 4.3). See FTP(S) Support and SFTP Support for more information.

J.6. Changes between 5.0 and 5.1

J.6.1. New Components

The following components are new in 5.1:

- `AmqpDedicatedChannelAdvice`

`AmqpDedicatedChannelAdvice`

See Strict Message Ordering.

Improved Function Support

The `java.util.function` interfaces now have improved integration support in the Framework components. Also Kotlin lambdas now can be used for handler and source methods.

See `java.util.function` Interfaces Support.

`@LongRunningTest`

A JUnit 5 `@LongRunningTest` conditional annotation is provided to check the environment or system properties for the `RUN_LONG_INTEGRATION_TESTS` entry with the value of `true` to determine if test should be run or skipped.

See JUnit Rules and Conditions.

J.6.2. General Changes

The following changes have been made in version 5.1:

- Java DSL
- Dispatcher Exceptions
- Global Channel Interceptors
- `ObjectToJsonTransformer`
- Integration Flows: Generated Bean Names
- Aggregator Changes
- @Publisher annotation changes
- Integration Graph Customization
- Integration Global Properties

- The `receiveTimeout` for `@Poller`

Java DSL

The `IntegrationFlowContext` is now an interface and `IntegrationFlowRegistration` is an inner interface of `IntegrationFlowContext`.

A new `logAndReply()` operator has been introduced for convenience when you wish to log at the end of a flow for request-reply configurations. This avoid confusion with `log()` which is treated as a one-way end flow component.

A generated bean name for any `NamedComponent` within an integration flow is now based on the component type for better readability from visual tools, logs analyzers and metrics collectors.

The `GenericHandler.handle()` now excepts a `MessageHeaders` type for the second argument.

Dispatcher Exceptions

Exceptions caught and re-thrown by `AbstractDispatcher` are now more consistent:

- A `MessagingException` of any kind that has a `failedMessage` property is re-thrown unchanged.
- All other exceptions are wrapped in a `MessageDeliveryException` with the `failedMessage` property set.

Previously:

- A `MessagingException` of any kind that has a `failedMessage` property was re-thrown unchanged
- A `MessagingException` that had no `failedMessage` property was wrapped in a `MessagingException` with the `failedMessage` property set.
- Other `RuntimeException` instances were re-thrown unchanged.
- Checked exceptions were wrapped in a `MessageDeliveryException` with the `failedMessage` property set.

Global Channel Interceptors

Global channel interceptors now apply to dynamically registered channels, such as through the `IntegrationFlowContext` when using the Java DSL or beans that are initialized using `beanFactory.initializeBean()`. Previously, when beans were created after the application context was refreshed, interceptors were not applied.

Channel Interceptors

`ChannelInterceptor.postReceive()` is no longer called when no message is received; it is no longer necessary to check for a `null Message<?>`. Previously, the method was called. If you have an interceptor that relies on the previous behavior, implement `afterReceiveCompleted()` instead, since that method is invoked, regardless of whether a message is received or not. Furthermore, the `PolledAmqpChannel` and `PolledJmsChannel` previously did not invoke `afterReceiveCompleted()` with `null`; they now do.

`ObjectToJsonTransformer`

A new `ResultType.BYTES` mode is introduced for the `ObjectToJsonTransformer`.

See JSON Transformers for more information.

Integration Flows: Generated Bean Names

Starting with version 5.0.5, generated bean names for the components in an `IntegrationFlow` include the flow bean name, followed by a dot, as a prefix. For example, if a flow bean were named `flowBean`, a generated bean might be named `flowBean.generatedBean`.

See Working With Message Flows for more information.

Aggregator Changes

If the `groupTimeout` is evaluated to a negative value, an aggregator now expires the group immediately. Only `null` is considered as a signal to do nothing for the current message.

A new `popSequence` property has been introduced to allow (by default) to call a `MessageBuilder.popSequenceDetails()` for the output message. Also an `AbstractAggregatingMessageGroupProcessor` returns now an `AbstractIntegrationMessageBuilder` instead of the whole `Message` for optimization.

See Aggregator for more information.

@Publisher annotation changes

Starting with version 5.1, you must explicitly turn on the `@Publisher` AOP functionality by using `@EnablePublisher` or by using the `<int:enable-publisher>` child element on `<int:annotation-config>`. Also the `proxy-target-class` and `order` attributes have been added for tuning the `ProxyFactory` configuration.

See Annotation-driven Configuration with the `@Publisher` Annotation for more information.

J.6.3. Files Changes

If you are using `FileExistsMode.APPEND` or `FileExistsMode.APPEND_NO_FLUSH` you can provide a `newFileCallback` that will be called when creating a new file. This callback receives the newly created file and the message that triggered the callback. This could be used to write a CSV header, for an example.

The `FileReadingMessageSource` now doesn't check and create a directory until its `start()` is called. So, if an Inbound Channel Adapter for the `FileReadingMessageSource` has `autoStartup = false`, there are no failures against the file system during application start up.

See File Support for more information.

J.6.4. AMQP Changes

We have made `ID` and `Timestamp` header mapping changes in the `DefaultAmqpHeaderMapper`. See the

note near the bottom of AMQP Message Headers for more information.

The `contentType` header is now correctly mapped as an entry in the general headers map. See contentType Header for more information.

Starting with version 5.1.3, if a message conversion exception occurs when using manual acknowledgments, and an error channel is defined, the payload is a `ManualAckListenerExecutionFailedException` with additional `channel` and `deliveryTag` properties. This enables the error flow to ack/nack the original message. See Inbound Message Conversion for more information.

J.6.5. JDBC Changes

A confusing `max-rows-per-poll` property on the JDBC Inbound Channel Adapter and JDBC Outbound Gateway has been deprecated in favor of the newly introduced `max-rows` property.

The `JdbcMessageHandler` supports now a `batchUpdate` functionality when the payload of the request message is an instance of an `Iterable` type.

The indexes for the `INT_CHANNEL_MESSAGE` table (for the `JdbcChannelMessageStore`) have been optimized. If you have large message groups in such a store, you may wish to alter the indexes.

See JDBC Support for more information.

J.6.6. FTP and SFTP Changes

A `RotatingServerAdvice` is now available to poll multiple servers and directories with the inbound channel adapters. See Inbound Channel Adapters: Polling Multiple Servers and Directories and Inbound Channel Adapters: Polling Multiple Servers and Directories for more information.

Also, inbound adapter `localFilenameExpression` instances can contain the `#remoteDirectory` variable, which contains the remote directory being polled. The generic type of the comparators (used to sort the fetched file list for the streaming adapters) has changed from `Comparator<AbstractFileInfo<F>>` to `Comparator<F>`. See FTP Streaming Inbound Channel Adapter and SFTP Streaming Inbound Channel Adapter for more information.

In addition, the synchronizers for inbound channel adapters can now be provided with a `Comparator`. This is useful when using `maxFetchSize` to limit the files retrieved.

The `CachingSessionFactory` has a new property `testSession` which, when true, causes the factory to perform a `test()` operation on the `Session` when checking out an existing session from the cache.

See SFTP Session Caching and FTP Session Caching for more information.

The outbound gateway MPUT command now supports a message payload with a collection of files or strings. See SFTP Outbound Gateway and FTP Outbound Gateway for more information.

J.6.7. TCP Support

When using SSL, host verification is now enabled, by default, to prevent man-in-the-middle attacks

with a trusted certificate. See Host Verification for more information.

In addition the key and trust store types can now be configured on the `DefaultTcpSSLContextSupport`.

J.6.8. Twitter Support

Since the Spring Social project has moved to end of life status, Twitter support in Spring Integration has been moved to the Extensions project. See Spring Integration Social Twitter for more information.

J.6.9. JMS Support

The `JmsSendingMessageHandler` now provides `deliveryModeExpression` and `timeToLiveExpression` options to determine respective QoS options for JMS message to send at runtime. The `DefaultJmsHeaderMapper` now allows to map inbound `JMSDeliveryMode` and `JMSExpiration` properties via setting to `true` respective `setMapInboundDeliveryMode()` and `setMapInboundExpiration()` options. When a `JmsMessageDrivenEndpoint` or `JmsInboundGateway` is stopped, the associated listener container is now shut down; this closes its shared connection and any consumers. You can configure the endpoints to revert to the previous behavior.

See JMS Support for more information.

J.6.10. HTTP/WebFlux Support

The `statusCodeExpression` (and `Function`) is now supplied with the `RequestEntity<?>` as a root object for evaluation context, so request headers, method, URI and body are available for target status code calculation.

See HTTP Support and WebFlux Support for more information.

J.6.11. JMX Changes

Object name key values are now quoted if they contain any characters other than those allowed in a Java identifier (or period `.`). For example `org.springframework.integration:type=MessageChannel,name="input:foo.myGroup.errors"`. This has the side effect that previously "allowed" names, with such characters, will now be quoted. For example `org.springframework.integration:type=MessageChannel,name="input#foo.myGroup.errors"`.

J.6.12. Micrometer Support Changes

It is now simpler to customize the standard Micrometer meters created by the framework. See Micrometer Integration for more information.

J.6.13. Integration Graph Customization

It is now possible to add additional properties to the `IntegrationNode` s via `Function<NamedComponent, Map<String, Object>> additionalPropertiesCallback` on the `IntegrationGraphServer`. See Integration Graph for more information.

J.6.14. Integration Global Properties

The Integration global properties (including defaults) can now be printed in the logs, when a `DEBUG` logic level is turned on for the `org.springframework.integration` category. See Global Properties for more information.

J.6.15. The `receiveTimeout` for `@Poller`

The `@Poller` annotation now provides a `receiveTimeout` option for convenience. See Using the `@Poller` Annotation for more information.

J.7. Changes between 4.3 and 5.0

See the Migration Guide for important changes that might affect your applications. You can find migration guides for all versions back to 2.1 on the wiki.

J.7.1. New Components

Version 5.0 added a number of new components.

Java DSL

The separate Spring Integration Java DSL project has now been merged into the core Spring Integration project. The `IntegrationComponentSpec` implementations for channel adapters and gateways are distributed to their specific modules. See Java DSL for more information about Java DSL support. See also the 4.3 to 5.0 Migration Guide for the required steps to move to Spring Integration 5.0.

Testing Support

We created a new Spring Integration Test Framework to help with testing Spring Integration applications. Now, with the `@SpringIntegrationTest` annotation on test classes and the `MockIntegration` factory, you can make your JUnit tests for integration flows somewhat easier.

See Testing support for more information.

MongoDB Outbound Gateway

The new `MongoDbOutboundGateway` lets you make queries to the database on demand by sending a message to its request channel.

See MongoDB Outbound Gateway for more information.

WebFlux Gateways and Channel Adapters

We introduced the new WebFlux support module for Spring WebFlux Framework gateways and channel adapters.

See WebFlux Support for more information.

Content Type Conversion

Now that we use the new `InvocableHandlerMethod`-based infrastructure for service method invocations, we can perform `contentType` conversion from the payload to a target method argument.

See Content Type Conversion for more information.

`ErrorMessagePublisher` and `ErrorMessageStrategy`

We added `ErrorMessagePublisher` and the `ErrorMessageStrategy` for creating `ErrorMessage` instances.

See Error Handling for more information.

JDBC Metadata Store

We added a JDBC implementation of the `MetadataStore` implementation. This is useful when you need to ensure transactional boundaries for metadata.

See JDBC Metadata Store for more information.

J.7.2. General Changes

Spring Integration is now fully based on Spring Framework `5.0` and Project Reactor `3.1`. Previous Project Reactor versions are no longer supported.

Core Changes

The `@Poller` annotation now has the `errorChannel` attribute for easier configuration of the underlying `MessagePublishingErrorHandler`. See Annotation Support for more information.

All the request-reply endpoints (based on `AbstractReplyProducingMessageHandler`) can now start transactions and, therefore, make the whole downstream flow transactional. See Transaction Support for more information.

The `SmartLifecycleRoleController` now provides methods to obtain status of endpoints in roles. See Endpoint Roles for more information.

By default, POJO methods are now invoked by using an `InvocableHandlerMethod`, but you can configure them to use SpEL, as before. See POJO Method invocation for more information.

When targeting POJO methods as message handlers, you can now mark one of the service methods with the `@Default` annotation to provide a fallback mechanism for non-matched conditions. See Configuring Service Activator for more information.

We added a simple `PassThroughTransactionSynchronizationFactory` to always store a polled message in the current transaction context. That message is used as a `failedMessage` property of the `MessagingException`, which wraps any raw exception thrown during transaction completion. See Transaction Synchronization for more information.

The aggregator expression-based `ReleaseStrategy` now evaluates the expression against the

`MessageGroup` instead of just the collection of `Message<?>`. See Aggregators and Spring Expression Language (SpEL) for more information.

You can now supply the `ObjectToMapTransformer` with a customized `JsonObjectMapper`.

See Aggregators and Spring Expression Language (SpEL) for more information.

The `@GlobalChannelInterceptor` annotation and `<int:channel-interceptor>` now support negative patterns (via `!` prepending) for component names matching. See Global Channel Interceptor Configuration for more information.

When a candidate failed to acquire the lock, the `LockRegistryLeaderInitiator` now emits a new `OnFailedToAcquireMutexEvent` through `DefaultLeaderEventPublisher`. See `Leadership Event Handling` for more information.

Gateway Changes

When the gateway method has a `void` return type and an error channel is provided, the gateway now correctly sets the `errorChannel` header. Previously, the header was not populated. This caused synchronous downstream flows (running on the calling thread) to send the exception to the configured channel, but an exception on an asynchronous downstream flow would be sent to the default `errorChannel` instead.

The `RequestReplyExchanger` interface now has a `throws MessagingException` clause to meet the proposed messages exchange contract.

You can now specify the request and reply timeouts with SpEL expressions. See Messaging Gateways for more information.

Aggregator Performance Changes

By default, aggregators now use a `SimpleSequenceSizeReleaseStrategy`, which is more efficient, especially with large groups. Empty groups are now scheduled for removal after `empty-group-min-timeout`. See Aggregator for more information.

Splitter Changes

The splitter component can now handle and split Java `Stream` and Reactive Streams `Publisher` objects. If the output channel is a `ReactiveStreamsSubscribableChannel`, the `AbstractMessageSplitter` builds a `Flux` for subsequent iteration instead of a regular `Iterator`, independent of the object being split. In addition, `AbstractMessageSplitter` provides `protected obtainSizeIfPossible()` methods to allow determination of the size of the `Iterable` and `Iterator` objects, if that is possible. See Splitter for more information.

JMS Changes

Previously, Spring Integration JMS XML configuration used a default bean name of `connectionFactory` for the JMS connection factory, letting the property be omitted from component definitions. We renamed it to `jmsConnectionFactory`, which is the bean name used by Spring Boot to auto-configure the JMS connection factory bean.

If your application relies on the previous behavior, you can rename your `connectionFactory` bean to `jmsConnectionFactory` or specifically configure your components to use your bean by using its current name. See JMS Support for more information.

Mail Changes

Some inconsistencies with rendering IMAP mail content have been resolved. See the note in the "Mail-receiving Channel Adapter" section for more information.

Feed Changes

Instead of the `com.rometools.fetcher.FeedFetcher`, which is deprecated in ROME, we introduced a new `Resource` property for the `FeedEntryMessageSource`. See Feed Adapter for more information.

File Changes

We introduced the new `FileHeaders.RELATIVE_PATH` message header to represent relative path in `FileReadingMessageSource`.

The tail adapter now supports `idleEventInterval` to emit events when there is no data in the file during that period.

The flush predicates for the `FileWritingMessageHandler` now have an additional parameter.

The file outbound channel adapter and gateway (`FileWritingMessageHandler`) now support the `REPLACE_IF_MODIFIED` `FileExistsMode`.

They also now support setting file permissions on the newly written file.

A new `FileSystemMarkerFilePresentFileListFilter` is now available. See Dealing With Incomplete Data for more information.

The `FileSplitter` now provides a `firstLineAsHeader` option to carry the first line of content as a header in the messages emitted for the remaining lines.

See File Support for more information.

FTP and SFTP Changes

The inbound channel adapters now have a property called `max-fetch-size`, which is used to limit the number of files fetched during a poll when no files are currently in the local directory. By default, they also are configured with a `FileSystemPersistentAcceptOnceFileListFilter` in the `local-filter`.

You can also provide a custom `DirectoryScanner` implementation to inbound channel adapters by setting the newly introduced `scanner` attribute.

You can now configure the regex and pattern filters to always pass directories. This can be useful when you use recursion in the outbound gateways.

By default, all the inbound channel adapters (streaming and synchronization-based) now use an

appropriate `AbstractPersistentAcceptOnceFileListFilter` implementation to prevent duplicate downloads of remote files.

The FTP and SFTP outbound gateways now support the `REPLACE_IF_MODIFIED FileExistsMode` when fetching remote files.

The FTP and SFTP streaming inbound channel adapters now add remote file information in a message header.

The FTP and SFTP outbound channel adapters (as well as the `PUT` command for outbound gateways) now support `InputStream` as `payload`, too.

The inbound channel adapters can now build file trees locally by using a newly introduced `RecursiveDirectoryScanner`. See the `scanner` option in the FTP Inbound Channel Adapter section for injection. Also, you can now switch these adapters to the `WatchService` instead.

We added The `NLST` command to the `AbstractRemoteFileOutboundGateway` to perform the list files names remote command.

You can now supply the `FtpOutboundGateway` with `workingDirExpression` to change the FTP client working directory for the current request message.

The `RemoteFileTemplate` is supplied now with the `invoke(OperationsCallback<F, T> action)` to perform several `RemoteFileOperations` calls in the scope of the same, thread-bounded, `Session`.

We added new filters for detecting incomplete remote files.

The `FtpOutboundGateway` and `SftpOutboundGateway` now support an option to remove the remote file after a successful transfer by using the `GET` or `MGET` commands.

See FTP/FTPS Adapters and SFTP Adapters for more information.

Integration Properties

Version 4.3.2 added a new `spring.integration.readOnly.headers` global property to let you customize the list of headers that should not be copied to a newly created `Message` by the `MessageBuilder`. See Global Properties for more information.

Stream Changes

We added a new option on the `CharacterStreamReadingMessageSource` to let it be used to “pipe” stdin and publish an application event when the pipe is closed. See Reading from Streams for more information.

Barrier Changes

The `BarrierMessageHandler` now supports a discard channel to which late-arriving trigger messages are sent. See Thread Barrier for more information.

AMQP Changes

The AMQP outbound endpoints now support setting a delay expression when you use the RabbitMQ Delayed Message Exchange plugin.

The inbound endpoints now support the Spring AMQP `DirectMessageListenerContainer`.

Pollable AMQP-backed channels now block the poller thread for the poller's configured `receiveTimeout` (default: one second).

Headers, such as `contentType`, that are added to message properties by the message converter are now used in the final message. Previously, it depended on the converter type as to which headers and message properties appeared in the final message. To override the headers set by the converter, set the `headersMappedLast` property to `true`. See AMQP Support for more information.

HTTP Changes

By default, the `DefaultHttpHeaderMapper.userDefinedHeaderPrefix` property is now an empty string instead of `X-`. See HTTP Header Mappings for more information.

By default, `uriVariablesExpression` now uses a `SimpleEvaluationContext` (since 5.0.4).

See Mapping URI Variables for more information.

MQTT Changes

Inbound messages are now mapped with the `RECEIVED_TOPIC`, `RECEIVED_QOS`, and `RECEIVED_RETAINED` headers to avoid inadvertent propagation to outbound messages when an application relays messages.

The outbound channel adapter now supports expressions for the topic, qos, and retained properties. The defaults remain the same. See MQTT Support for more information.

STOMP Changes

We changed the STOMP module to use `ReactorNettyTcpStompClient`, based on the Project Reactor `3.1` and `reactor-netty` extension. We renamed `Reactor2TcpStompSessionManager` to `ReactorNettyTcpStompSessionManager`, according to the `ReactorNettyTcpStompClient` foundation. See STOMP Support for more information.

Web Services Changes

You can now supply `WebServiceOutboundGateway` instances with an externally configured `WebServiceTemplate` instances.

`DefaultSoapHeaderMapper` can now map a `javax.xml.transform.Source` user-defined header to a SOAP header element.

Simple WebService inbound and outbound gateways can now deal with the complete `WebServiceMessage` as a `payload`, allowing the manipulation of MTOM attachments.

See Web Services Support for more information.

Redis Changes

The `RedisStoreWritingMessageHandler` is supplied now with additional `String`-based setters for SpEL expressions (for convenience with Java configuration). You can now configure the `zsetIncrementExpression` on the `RedisStoreWritingMessageHandler` as well. In addition, this property has been changed from `true` to `false` since the `INCR` option on `ZADD` Redis command is optional.

You can now supply the `RedisInboundChannelAdapter` with an `Executor` for executing Redis listener invokers. In addition, the received messages now contain a `RedisHeaders.MESSAGE_SOURCE` header to indicate the source of the message (topic or pattern).

See Redis Support for more information.

TCP Changes

We added a new `ThreadAffinityClientConnectionFactory` to bind TCP connections to threads.

You can now configure the TCP connection factories to support `PushbackInputStream` instances, letting deserializers “unread” (push back) bytes after “reading ahead”.

We added a `ByteArrayElasticRawDeserializer` without `maxMessageSize` to control and buffer incoming data as needed.

See TCP and UDP Support for more information.

Gemfire Changes

The `GemfireMetadataStore` now implements `ListenableMetadataStore`, letting you listen to cache events by providing `MetadataStoreListener` instances to the store. See Pivotal GemFire and Apache Geode Support for more information.

JDBC Changes

The `JdbcMessageChannelStore` now provides a setter for `ChannelMessageStorePreparedStatementSetter`, letting you customize message insertion in the store.

The `ExpressionEvaluatingSqlParameterSourceFactory` now provides a setter for `sqlParameterTypes`, letting you customize the SQL types of the parameters.

See JDBC Support for more information.

Metrics Changes

Micrometer application monitoring is now supported (since version 5.0.2). See Micrometer Integration for more information.

Changes were made to the Micrometer `Meters` in version 5.0.3 to make them more suitable for use in dimensional systems. Further changes were made in 5.0.4. If you use Micrometer, we recommend a minimum of version 5.0.4.

`@EndpointId` Annotations

Introduced in version 5.0.4, this annotation provides control over bean naming when you use Java configuration. See Endpoint Bean Names for more information.

J.8. Changes between 4.2 and 4.3

See the Migration Guide for important changes that might affect your applications. You can find migration guides for all versions back to 2.1 on the Wiki.

J.8.1. New Components

Version 4.3 added a number of new components.

AMQP Async Outbound Gateway

See Asynchronous Outbound Gateway.

`MessageGroupFactory`

We introduced the `MessageGroupFactory` strategy to allow control over `MessageGroup` instances in `MessageGroupStore` logic. We added `SimpleMessageGroupFactory` implementation for the `SimpleMessageGroup`, with the `GroupType.HASH_SET` as the default factory for the standard `MessageGroupStore` implementations. See Message Store for more information.

`PersistentMessageGroup`

We added the `PersistentMessageGroup` (lazy-load proxy) implementation for persistent `MessageGroupStore` instances, which return this instance for the `getMessageGroup()` when their `lazyLoadMessageGroups` is `true` (the default). See Message Store for more information.

FTP and SFTP Streaming Inbound Channel Adapters

We added inbound channel adapters that return an `InputStream` for each file, letting you retrieve remote files without writing them to the local file system. See FTP Streaming Inbound Channel Adapter and SFTP Streaming Inbound Channel Adapter for more information.

`StreamTransformer`

We added `StreamTransformer` to transform an `InputStream` payload to either a `byte[]` or a `String`. See Stream Transformer for more information.

Integration Graph

We added `IntegrationGraphServer`, together with the `IntegrationGraphController` REST service, to expose the runtime model of a Spring Integration application as a graph. See Integration Graph for more information.

JDBC Lock Registry

We added `JdbcLockRegistry` for distributed locks shared through a database table. See JDBC Lock

Registry for more information.

`LeaderInitiator` for `LockRegistry`

We added `LeaderInitiator` implementation based on the `LockRegistry` strategy. See Leadership Event Handling for more information.

J.8.2. General Changes

This section describes general changes that version 4.3 brought to Spring Integration.

Core Changes

This section describes general changes to the core of Spring Integration.

Outbound Gateway within a Chain

Previously, you could specify a `reply-channel` on an outbound gateway within a chain. It was completely ignored. The gateway's reply goes to the next chain element or, if the gateway is the last element, to the chain's output channel. This condition is now detected and disallowed. If you have such a configuration, remove the `reply-channel`.

Asynchronous Service Activator

We added an option to make the service activator be synchronous. See Asynchronous Service Activator for more information.

Messaging Annotation Support changes

The messaging annotation support does not require a `@MessageEndpoint` (or any other `@Component`) annotation declaration on the class level. To restore the previous behavior, set the `spring.integration.messagingAnnotations.require.componentAnnotation` of `spring.integration.properties` to `true`. See Global Properties and Annotation Support for more information.

Mail Changes

This section describes general changes to the Spring Integration Mail functionality.

Customizable User Flag

The customizable `userFlag` (added in 4.2.2 to provide customization of the flag used to denote that the mail has been seen) is now available in the XML namespace. See Marking IMAP Messages When `\Recent` Is Not Supported for more information.

Mail Message Mapping

You can now map inbound mail messages with the `MessageHeaders` containing the mail headers and the payload containing the email content. Previously, the payload was always the raw `MimeMessage`. See Inbound Mail Message Mapping for more information.

JMS Changes

This section describes general changes to the Spring Integration JMS functionality.

Header Mapper

The `DefaultJmsHeaderMapper` now maps the standard `correlationId` header as a message property by invoking its `toString()` method. See Mapping Message Headers to and from JMS Message for more information.

Asynchronous Gateway

The JMS outbound gateway now has an `async` property. See Async Gateway for more information.

Aggregator Changes

There is a change in behavior when a POJO aggregator releases a collection of `Message<?>` objects. This is rare, but, if your application does that, you need to make a small change to your POJO. See this IMPORTANT: The `SimpleMessageGroup.getMessages()` method returns an `unmodifiableCollection`. note for more information.

TCP/UDP Changes

This section describes general changes to the Spring Integration TCP/UDP functionality.

Events

A new `TcpConnectionServerListeningEvent` is emitted when a server connection factory is started. See TCP Connection Events for more information.

You can now use the `destination-expression` and `socket-expression` attributes on `<int-ip:udp-outbound-channel-adapter>`. See UDP Adapters for more information.

Stream Deserializers

The various deserializers that cannot allocate the final buffer until the whole message has been assembled now support pooling the raw buffer into which the data is received rather than creating and discarding a buffer for each message. See TCP Connection Factories for more information.

TCP Message Mapper

The message mapper now, optionally, sets a configured content type header. See IP Message Headers for more information.

File Changes

This section describes general changes to the Spring Integration File functionality.

Destination Directory Creation

The generated file name for the `FileWritingMessageHandler` can represent a sub-path to save the desired directory structure for a file in the target directory. See Generating File Names for more

information.

The `FileReadingMessageSource` now hides the `WatchService` directory scanning logic in the inner class. We added the `use-watch-service` and `watch-events` options to enable this behavior. We deprecated the top-level `WatchServiceDirectoryScanner` because of inconsistency around the API. See `WatchServiceDirectoryScanner` for more information.

Buffer Size

When writing files, you can now specify the buffer size.

Appending and Flushing

You can now avoid flushing files when appending and use a number of strategies to flush the data during idle periods. See Flushing Files When Using `APPEND_NO_FLUSH` for more information.

Preserving Timestamps

You can now configure the outbound channel adapter to set the destination file's `lastmodified` timestamp. See File Timestamps for more information.

Splitter Changes

The `FileSplitter` now automatically closes an FTP or SFTP session when the file is completely read. This applies when the outbound gateway returns an `InputStream` or when you use the new FTP or SFTP streaming channel adapters. We also introduced a new `markers-json` option to convert `FileSplitter.FileMarker` to JSON `String` for relaxed downstream network interaction. See File Splitter for more information.

File Filters

We added `ChainFileListFilter` as an alternative to `CompositeFileListFilter`. See Reading Files for more information.

AMQP Changes

This section describes general changes to the Spring Integration AMQP functionality.

Content Type Message Converter

The outbound endpoints now support a `RabbitTemplate` configured with a `ContentTypeDelegatingMessageConverter` such that you can choose the converter based on the message content type. See Outbound Message Conversion for more information.

Headers for Delayed Message Handling

Spring AMQP 1.6 adds support for delayed message exchanges. Header mapping now supports the headers (`amqp_delay` and `amqp_receivedDelay`) used by this feature.

AMQP-Backed Channels

AMQP-backed channels now support message mapping. See AMQP-backed Message Channels for

more information.

Redis Changes

This section describes general changes to the Spring Integration Redis functionality.

List Push/Pop Direction

Previously, the queue channel adapters always used the Redis list in a fixed direction, pushing to the left end and reading from the right end. You can now configure the reading and writing direction with the `rightPop` and `leftPush` options for the `RedisQueueMessageDrivenEndpoint` and `RedisQueueOutboundChannelAdapter`, respectively. See Redis Queue Inbound Channel Adapter and Redis Queue Outbound Channel Adapter for more information.

Queue Inbound Gateway Default Serializer

The default serializer in the inbound gateway has been changed to a `JdkSerializationRedisSerializer` for compatibility with the outbound gateway. See Redis Queue Inbound Gateway for more information.

HTTP Changes

Previously, with requests that had a body (such as `POST`) that had no `content-type` header, the body was ignored. With this release, the content type of such requests is considered to be `application/octet-stream` as recommended by RFC 2616. See Http Inbound Components for more information.

`uriVariablesExpression` now uses a `SimpleEvaluationContext` by default (since 4.3.15). See Mapping URI Variables for more information.

SFTP Changes

This section describes general changes to the Spring Integration SFTP functionality.

Factory Bean

We added a new factory bean to simplify the configuration of Jsch proxies for SFTP. See Proxy Factory Bean for more information.

`chmod` Changes

The SFTP outbound gateway (for `put` and `mput` commands) and the SFTP outbound channel adapter now support the `chmod` attribute to change the remote file permissions after uploading. See `SFTP Outbound Channel Adapter` and `SFTP Outbound Gateway` for more information.

FTP Changes

This section describes general changes to the Spring Integration FTP functionality.

Session Changes

The `FtpSession` now supports `null` for the `list()` and `listNames()` methods, since underlying FTP Client can use it. With that, you can now configure the `FtpOutboundGateway` without the `remoteDirectory` expression. You can also configure the `<int-ftp:inbound-channel-adapter>` without `remote-directory` or `remote-directory-expression`. See FTP/FTPS Adapters for more information.

Router Changes

The `ErrorMessageExceptionTypeRouter` now supports the `Exception` superclass mappings to avoid duplication for the same channel in case of multiple inheritors. For this purpose, the `ErrorMessageExceptionTypeRouter` loads mapping classes during initialization to fail-fast for a `ClassNotFoundException`.

See Routers for more information.

Header Mapping

This section describes the changes to header mapping between version 4.2 and 4.3.

General

AMQP, WS, and XMPP header mappings (such as `request-header-mapping` and `reply-header-mapping`) now support negated patterns. See AMQP Message Headers, WS Message Headers, and XMPP Message Headers for more information.

AMQP Header Mapping

Previously, only standard AMQP headers were mapped by default. You had to explicitly enable mapping of user-defined headers. With this release, all headers are mapped by default. In addition, the inbound `amqp_deliveryMode` header is no longer mapped by default. See AMQP Message Headers for more information.

Groovy Scripts

You can now configure groovy scripts with the `compile-static` hint or any other `CompilerConfiguration` options. See Groovy Configuration for more information.

`@InboundChannelAdapter` Changes

The `@InboundChannelAdapter` now has an alias `channel` attribute for the regular `value`. In addition, the target `SourcePollingChannelAdapter` components can now resolve the target `outputChannel` bean from its provided name (`outputChannelName` options) in a late-binding manner. See Annotation Support for more information.

XMPP Changes

The XMPP channel adapters now support the XMPP Extensions (XEP). See XMPP Extensions for more information.

WireTap Late Binding

The `WireTap ChannelInterceptor` now can accept a `channelName` that is resolved to the target `MessageChannel` later, during the first active interceptor operation. See Wire Tap for more information.

`ChannelMessageStoreQueryProvider` Changes

The `ChannelMessageStoreQueryProvider` now supports H2 databases. See Backing Message Channels for more information.

WebSocket Changes

The `ServerWebSocketContainer` now exposes an `allowedOrigins` option, and `SockJsServiceOptions` exposes a `suppressCors` option. See WebSockets Support for more information.

J.9. Changes between 4.1 and 4.2

See the Migration Guide for important changes that might affect your applications. You can find migration guides for all versions back to 2.1 on the wiki.

J.9.1. New Components

Version 4.2 added a number of new components.

Major Management/JMX Rework

We added a new `MetricsFactory` strategy interface. This change, together with other changes in the JMX and management infrastructure, provides much more control over management configuration and runtime performance.

However, this has some important implications for (some) user environments.

For complete details, see Metrics and Management and JMX Improvements.

MongoDB Metadata Store

The `MongoDbMetadataStore` is now available. For more information, see MongoDB Metadata Store.

SecuredChannel Annotation

We introduced the `@SecuredChannel` annotation, replacing the deprecated `ChannelSecurityInterceptorFactoryBean`. For more information, see Security in Spring Integration.

`SecurityContext` Propagation

We introduced the `SecurityContextPropagationChannelInterceptor` for the `SecurityContext` propagation from one message flow's thread to another. For more information, see Security in Spring Integration.

FileSplitter

In 4.1.2, we added `FileSplitter`, which splits text files into lines. It now has full support in the `int-file:` namespace. See File Splitter for more information.

Zookeeper Support

We added Zookeeper support to the framework to assist when running on a clustered or multi-host environment. The change impacts the following features:

- `ZookeeperMetadataStore`
- `ZookeeperLockRegistry`
- Zookeeper Leadership

See Zookeeper Support for more information.

Thread Barrier

A new thread `<int:barrier/>` component is available, letting a thread be suspended until some asynchronous event occurs. See Thread Barrier for more information.

STOMP Support

We added STOMP support to the framework as an inbound and outbound channel adapters pair. See STOMP Support for more information.

Codec

A new `Codec` abstraction has been introduced, to encode and decode objects to and from `byte[]`. We added an implementation that uses Kryo. We also added codec-based transformers and message converters. See Codec for more information.

Message PreparedStatement Setter

A new `MessagePreparedStatementSetter` functional interface callback is available for the `JdbcMessageHandler` (`<int-jdbc:outbound-gateway>` and `<int-jdbc:outbound-channel-adapter>`) as an alternative to using `SqlParameterSourceFactory` to populate parameters on the `PreparedStatement` with the `requestMessage` context. See Outbound Channel Adapter for more information.

J.9.2. General Changes

This section describes general changes from version 4.1 to version 4.2.

WireTap

As an alternative to the existing `selector` attribute, the `<wire-tap/>` element now supports the `selector-expression` attribute.

File Changes

See File Support for more information about these changes.

Appending New Lines

The `<int-file:outbound-channel-adapter>` and `<int-file:outbound-gateway>` now support an `append-new-line` attribute. If set to `true`, a new line is appended to the file after a message is written. The default attribute value is `false`.

Ignoring Hidden Files

We added the `ignore-hidden` attribute for the `<int-file:inbound-channel-adapter>` to let you set whether to pick up hidden files from the source directory. It defaults to `true`.

Writing `InputStream` Payloads

The `FileWritingMessageHandler` now also accepts `InputStream` as a valid message payload type.

`HeadDirectoryScanner`

You can now use the `HeadDirectoryScanner` with other `FileListFilter` implementations.

Last Modified Filter

We added the `LastModifiedFileListFilter`.

Watch Service Directory Scanner

We added the `WatchServiceDirectoryScanner`.

Persistent File List Filter Changes

The `AbstractPersistentFileListFilter` has a new property (`flushOnUpdate`) which, when set to `true`, calls `flush()` on the metadata store if it implements `Flushable` (for example, `PropertiesPersistingMetadataStore`).

Class Package Change

We moved the `ScatterGatherHandler` class from the `org.springframework.integration.handler` to the `org.springframework.integration.scattergather`.

TCP Changes

This section describes general changes to the Spring Integration TCP functionality.

TCP Serializers

The TCP `Serializers` no longer `flush()` the `OutputStream`. This is now done by the `TcpNxxConnection` classes. If you use the serializers directly within your code, you may have to `flush()` the `OutputStream`.

Server Socket Exceptions

`TcpConnectionServerExceptionEvent` instances are now published whenever an unexpected exception occurs on a TCP server socket (also added to 4.1.3 and 4.0.7). See TCP Connection Events for more information.

TCP Server Port

If you configure a TCP server socket factory to listen on a random port, you can now obtain the actual port chosen by the OS by using `getPort()`. `getServerSocketAddress()` is also available.

See "TCP Connection Factories" for more information.

TCP Gateway Remote Timeout

The `TcpOutboundGateway` now supports `remote-timeout-expression` as an alternative to the existing `remote-timeout` attribute. This allows setting the timeout based on each message.

Also, the `remote-timeout` no longer defaults to the same value as `reply-timeout`, which has a completely different meaning.

See .TCP Outbound Gateway Attributes for more information.

TCP SSLSession Available for Header Mapping

`TcpConnection` implementations now support `getSslSession()` to let you extract information from the session to add to message headers. See IP Message Headers for more information.

TCP Events

New events are now published whenever a correlation exception occurs — such as sending a message to a non-existent socket.

The `TcpConnectionEventListeningMessageProducer` is deprecated. Use the generic event adapter instead.

See TCP Connection Events for more information.

`@InboundChannelAdapter` Changes

Previously, the `@Poller` on an inbound channel adapter defaulted the `maxMessagesPerPoll` attribute to `-1` (infinity). This was inconsistent with the XML configuration of `<inbound-channel-adapter/>`, which defaults to `1`. The annotation now defaults this attribute to `1`.

API Changes

`o.s.integration.util.FunctionIterator` now requires a `o.s.integration.util.Function` instead of a `reactor.function.Function`. This was done to remove an unnecessary hard dependency on Reactor. Any uses of this iterator need to change the import.

Reactor is still supported for functionality such as the `Promise` gateway. The dependency was removed for those users who do not need it.

JMS Changes

This section describes general changes to the Spring Integration TCP functionality.

Reply Listener Lazy Initialization

You can now configure the reply listener in JMS outbound gateways to be initialized on-demand and stopped after an idle period, instead of being controlled by the gateway's lifecycle. See Outbound Gateway for more information.

Conversion Errors in Message-Driven Endpoints

The `error-channel` is now used for the conversion errors. In previous versions, they caused transaction rollback and message redelivery.

See Message-driven Channel Adapter and Inbound Gateway for more information.

Default Acknowledge Mode

When using an implicitly defined `DefaultMessageListenerContainer`, the default `acknowledge` is now `transacted`. We recommend using `transacted` when using this container, to avoid message loss. This default now applies to the message-driven inbound adapter and the inbound gateway. It was already the default for JMS-backed channels.

See Message-driven Channel Adapter and Inbound Gateway for more information.

Shared Subscriptions

We added Namespace support for shared subscriptions (JMS 2.0) to message-driven endpoints and the `<int-jms:publish-subscribe-channel>`. Previously, you had to wire up listener containers as `<bean/>` declarations to use shared connections.

See JMS Support for more information.

Conditional Pollers

We now provide much more flexibility for dynamic polling.

See Conditional Pollers for Message Sources for more information.

AMQP Changes

This section describes general changes to the Spring Integration AMQP functionality.

Publisher Confirmations

The `<int-amqp:outbound-gateway>` now supports `confirm-correlation-expression`, `confirm-ack-channel`, and `confirm-nack-channel` attributes (which have a purpose similar to that of `<int-amqp:outbound-channel-adapter>`).

Correlation Data

For both the outbound channel adapter and the inbound gateway, if the correlation data is a `Message<?>`, it becomes the basis of the message on the ack or nack channel, with the additional header(s) added. Previously, any correlation data (including `Message<?>`) was returned as the payload of the ack or nack message.

Inbound Gateway Properties

The `<int-amqp:inbound-gateway>` now exposes the `amqp-template` attribute to allow more control over an external bean for the reply `RabbitTemplate`. You can also provide your own `AmqpTemplate` implementation. In addition, you can use `default-reply-to` if the request message does not have a `replyTo` property.

See AMQP Support for more information.

XPath Splitter Improvements

The `XPathMessageSplitter` (`<int-xml:xpath-splitter>`) now allows the configuration of `output-properties` for the internal `javax.xml.transform.Transformer` and supports an `Iterator` mode (defaults to `true`) for the XPath evaluation `org.w3c.dom.NodeList` result.

See Splitting XML Messages for more information.

HTTP Changes

This section describes general changes to the Spring Integration HTTP functionality.

CORS

The HTTP inbound endpoints (`<int-http:inbound-channel-adapter>` and `<int-http:inbound-gateway>`) now allow the configuration of Cross-origin Resource Sharing (CORS).

See Cross-origin Resource Sharing (CORS) Support for more information.

Inbound Gateway Timeout

You can configure the HTTP inbound gate way to return a status code that you specify when a request times out. The default is now `500 Internal Server Error` instead of `200 OK`.

See Response Status Code for more information.

Form Data

We added documentation for proxying `multipart/form-data` requests. See HTTP Support for more information.

Gateway Changes

This section describes general changes to the Spring Integration Gateway functionality.

Gateway Methods can Return `CompletableFuture<?>`

When using Java 8, gateway methods can now return `CompletableFuture<?>`. See `CompletableFuture` for more information.

MessagingGateway Annotation

The request and reply timeout properties are now `String` instead of `Long` to allow configuration with property placeholders or SpEL. See `@MessagingGateway` Annotation.

Aggregator Changes

This section describes general changes to the Spring Integration aggregator functionality.

Aggregator Performance

This release includes some performance improvements for aggregating components (aggregator, resequencer, and others), by more efficiently removing messages from groups when they are released. New methods (`removeMessagesFromGroup`) have been added to the message store. Set the `removeBatchSize` property (default: `100`) to adjust the number of messages deleted in each operation. Currently, the JDBC, Redis, and MongoDB message stores support this property.

Output Message Group Processor

When using a `ref` or inner bean for the aggregator, you can now directly bind a `MessageGroupProcessor`. In addition, we added a `SimpleMessageGroupProcessor` that returns the collection of messages in the group. When an output processor produces a collection of `Message<?>`, the aggregator releases those messages individually. Configuring the `SimpleMessageGroupProcessor` makes the aggregator a message barrier, where messages are held up until they all arrive and are then released individually. See Aggregator for more information.

FTP and SFTP Changes

This section describes general changes to the Spring Integration FTP and SFTP functionality.

Inbound Channel Adapters

You can now specify a `remote-directory-expression` on the inbound channel adapters, to determine the directory at runtime. See FTP/FTPS Adapters and SFTP Adapters for more information.

Gateway Partial Results

When you use FTP or SFTP outbound gateways to operate on multiple files (with `mget` and `mput`), an exception can occur after part of the request is completed. If such a condition occurs, a `PartialSuccessException` that contains the partial results is thrown. See FTP Outbound Gateway and SFTP Outbound Gateway for more information.

Delegating Session Factory

We added a delegating session factory, enabling the selection of a particular session factory based on some thread context value.

See Delegating Session Factory and Delegating Session Factory for more information.

Default Sftp Session Factory

Previously, the `DefaultSftpSessionFactory` unconditionally allowed connections to unknown hosts. This is now configurable (default: `false`).

The factory now requires a configured `knownHosts`, file unless the `allowUnknownKeys` property is `true` (default: `false`).

See `allowUnknownKeys`::Set to `true` to allow connections to hosts with unknown (or changed) keys. for more information.

Message Session Callback

We introduced the `MessageSessionCallback<F, T>` to perform any custom `Session` operations with the `requestMessage` context in the `<int-(s)ftp:outbound-gateway/>`.

See Using `MessageSessionCallback` and MessageSessionCallback for more information.

Websocket Changes

We added `WebSocketHandlerDecoratorFactory` support to the `ServerWebSocketContainer` to allow chained customization for the internal `WebSocketHandler`. See WebSockets Namespace Support for more information.

Application Event Adapters changes

The `ApplicationEvent` adapters can now operate with `payload` as an `event` to directly allow omitting custom `ApplicationEvent` extensions. For this purpose, we introduced the `publish-payload` boolean attribute has been introduced on the `<int-event:outbound-channel-adapter>`. See Spring `ApplicationEvent` Support for more information.

J.10. Changes between 4.0 and 4.1

See the Migration Guide for important changes that might affect your applications. You can find migration guides for all versions back to 2.1 on the wiki.

J.10.1. New Components

Version 4.1 added a number of new components.

Promise<?> Gateway

The messaging gateway methods now support a Reactor `Promise` return type. See Asynchronous Gateway.

WebSocket support

The `WebSocket` module is now available. It is fully based on the Spring WebSocket and Spring

Messaging modules and provides an `<inbound-channel-adapter>` and an `<outbound-channel-adapter>`. See WebSockets Support for more information.

Scatter-Gather Enterprise Integration Pattern

We implemented the scatter-gather enterprise integration pattern. See Scatter-Gather for more information.

Routing Slip Pattern

We added the routing slip EIP pattern implementation. See Routing Slip for more information.

Idempotent Receiver Pattern

We added the idempotent receiver enterprise integration pattern implementation by adding the `<idempotent-receiver>` component in XML or the `IdempotentReceiverInterceptor` and `IdempotentReceiver` annotations for Java configuration. See Idempotent Receiver Enterprise Integration Pattern and the Javadoc for more information.

Boon `JsonObjectMapper`

We added the Boon `JsonObjectMapper` for the JSON transformers. See Transformer for more information.

Redis Queue Gateways

We added the `<redis-queue-inbound-gateway>` and `<redis-queue-outbound-gateway>` components. See Redis Queue Inbound Gateway and Redis Queue Outbound Gateway.

`PollSkipAdvice`

We added the `PollSkipAdvice`, which you can use within the `<advice-chain>` of the `<poller>` to determine if the current poll should be suppressed (skipped) by some condition that you implement with `PollSkipStrategy`. See Poller for more information.

J.10.2. General Changes

This section describes general changes from version 4.0 to version 4.1.

AMQP Inbound Endpoints, Channel

Elements that use a message listener container (inbound endpoints and channel) now support the `missing-queues-fatal` attribute. See AMQP Support for more information.

AMQP Outbound Endpoints

The AMQP outbound endpoints support a new property called `lazy-connect` (default: `true`). When `true`, the connection to the broker is not established until the first message arrives (assuming there are no inbound endpoints, which always try to establish the connection during startup). When set to `false`, an attempt to establish the connection is made during application startup. See AMQP Support for more information.

SimpleMessageStore

The `SimpleMessageStore` no longer makes a copy of the group when calling `getMessageGroup()`. See [WARNING] for more information.

Web Service Outbound Gateway: `encode-uri`

The `<ws:outbound-gateway/>` now provides an `encode-uri` attribute to allow disabling the encoding of the URI object before sending the request.

Http Inbound Channel Adapter and Status Code

The `<http:inbound-channel-adapter>` can now be configured with a `status-code-expression` to override the default `200 OK` status. See HTTP Namespace Support for more information.

MQTT Adapter Changes

You can now configure the MQTT channel adapters to connect to multiple servers — for example, to support High Availability (HA). See MQTT Support for more information.

The MQTT message-driven channel adapter now supports specifying the QoS setting for each subscription. See Inbound (Message-driven) Channel Adapter for more information.

The MQTT outbound channel adapter now supports asynchronous sends, avoiding blocking until delivery is confirmed. See Outbound Channel Adapter for more information.

It is now possible to programmatically subscribe to and unsubscribe from topics at runtime. See Inbound (Message-driven) Channel Adapter for more information.

FTP and SFTP Adapter Changes

The FTP and SFTP outbound channel adapters now support appending to remote files and taking specific actions when a remote file already exists. The remote file templates now also supports this, as well as `rmdir()` and `exists()`. In addition, the remote file templates provide access to the underlying client object, enabling access to low-level APIs.

See FTP/FTPS Adapters and SFTP Adapters for more information.

Splitter and Iterator

`Splitter` components now support an `Iterator` as the result object for producing output messages. See Splitter for more information.

Aggregator

`Aggregator` instancess now support a new attribute `expire-groups-upon-timeout`. See Aggregator for more information.

Content Enricher Improvements

We added a `null-result-expression` attribute, which is evaluated and returned if `<enricher>` returns

`null`. You can add it in `<header>` and `<property>`. See Content Enricher for more information.

We added an `error-channel` attribute, which is used to handle an error flow if an `Exception` occurs downstream of the `request-channel`. This lets you return an alternative object to use for enrichment. See Content Enricher for more information.

Header Channel Registry

The `<header-enricher/>` element's `<header-channels-to-string/>` child element can now override the header channel registry's default time for retaining channel mappings. See Header Channel Registry for more information.

Orderly Shutdown

We made improvements to the orderly shutdown algorithm. See Orderly Shutdown for more information.

Management for `RecipientListRouter`

The `RecipientListRouter` now provides several management operations to configure recipients at runtime. With that, you can now configure the `<recipient-list-router>` without any `<recipient>` from the start. See `RecipientListRouterManagement` for more information.

AbstractHeaderMapper: NON_STANDARD_HEADERS token

The `AbstractHeaderMapper` implementation now provides the additional `NON_STANDARD_HEADERS` token to map any user-defined headers, which are not mapped by default. See AMQP Message Headers for more information.

AMQP Channels: `template-channel-transacted`

We introduced the `template-channel-transacted` attribute for AMQP `MessageChannel` instances. See AMQP-backed Message Channels for more information.

Syslog Adapter

The default syslog message converter now has an option to retain the original message in the payload while still setting the headers. See Syslog Inbound Channel Adapter for more information.

Asynchronous Gateway

In addition to the `Promise` return type mentioned earlier, gateway methods may now return a `ListenableFuture`, introduced in Spring Framework 4.0. You can also disable asynchronous processing in the gateway, letting a downstream flow directly return a `Future`. See Asynchronous Gateway.

Aggregator Advice Chain

`Aggregator` and `Resequencer` now support `<expire-advice-chain/>` and `<expire-transactional/>` child elements to advise the `forceComplete` operation. See Configuring an Aggregator with XML for more information.

Outbound Channel Adapter and Scripts

The `<int:outbound-channel-adapter/>` now supports the `<script/>` child element. The underlying script must have a `void` return type or return `null`. See Groovy support and Scripting Support.

Resequencer Changes

When a message group in a resequencer times out (using `group-timeout` or a `MessageGroupStoreReaper`), late arriving messages are now, by default, discarded immediately. See Resequencer.

Optional POJO method parameter

Spring Integration now consistently handles the Java 8's `Optional` type. See Configuring Service Activator.

`QueueChannel` backed Queue type

The `QueueChannel` backed `Queue type` has been changed from `BlockingQueue` to the more generic `Queue`. This change allows the use of any external `Queue` implementation (for example, Reactor's `PersistentQueue`). See `QueueChannel` Configuration.

`ChannelInterceptor` Changes

The `ChannelInterceptor` now supports additional `afterSendCompletion()` and `afterReceiveCompletion()` methods. See Channel Interceptors.

IMAP PEEK

Since version 4.1.1 there is a change of behavior if you explicitly set the `mail.[protocol].peek` JavaMail property to `false` (where `[protocol]` is `imap` or `imaps`). See [IMPORTANT].

J.11. Changes between 3.0 and 4.0

See the Migration Guide for important changes that might affect your applications. You can find migration guides for all versions back to 2.1 on the wiki.

J.11.1. New Components

Version 4.0 added a number of new components.

MQTT Channel Adapters

The MQTT channel adapters (previously available in the Spring Integration Extensions repository) are now available as part of the normal Spring Integration distribution. See MQTT Support.

`@EnableIntegration`

We added the `@EnableIntegration` annotation to permit declaration of standard Spring Integration beans when using `@Configuration` classes. See Annotation Support for more information.

`@IntegrationComponentScan`

We added the `@IntegrationComponentScan` annotation to permit classpath scanning for Spring Integration-specific components. See Annotation Support for more information.

"@EnableMessageHistory"

You can now enable message history with the `@EnableMessageHistory` annotation in a `@Configuration` class. In addition, a JMX MBean can modify the message history settings. Also, `MessageHistory` can track auto-created `MessageHandler` instances for annotated endpoints (such as `@ServiceActivator`, `@Splitter`, and others). For more information, see Message History.

`@MessagingGateway`

You can now configure messaging gateway interfaces with the `@MessagingGateway` annotation. It is an analogue of the `<int:gateway/>` XML element. For more information, see `@MessagingGateway` Annotation.

Spring Boot `@EnableAutoConfiguration`

As well as the `@EnableIntegration` annotation mentioned earlier, we introduced a hook to allow the Spring Integration infrastructure beans to be configured with Spring Boot's `@EnableAutoConfiguration` annotation. For more information, see "Auto-configuration" in the Spring Boot Reference Guide.

`@GlobalChannelInterceptor`

As well as the `@EnableIntegration` annotation mentioned above, we introduced the `@GlobalChannelInterceptor` annotation. For more information, see Annotation Support.

`@IntegrationConverter`

We introduced the `@IntegrationConverter` annotation as an analogue of the `<int:converter/>` component. For more information, see Annotation Support.

`@EnablePublisher`

We added the `@EnablePublisher` annotation to allow the specification of a `default-publisher-channel` for `@Publisher` annotations. See Annotation Support for more information.

Redis Channel Message Stores

We added a Redis `MessageGroupStore` that is optimized for use when backing a `QueueChannel` for persistence. For more information, see Redis Channel Message Stores.

We added a Redis `ChannelPriorityMessageStore`. You can use it to retrieve messages by priority. For more information, see Redis Channel Message Stores.

MongodDB Channel Message Store

The MongoDB support now provides the `MongoDbChannelMessageStore`, which is a channel-specific `MessageStore` implementation. With `priorityEnabled = true`, you can use it in `<int:priority-queue>`

elements to achieve priority order polling of persisted messages. For more information see MongoDB Channel Message Store.

`@EnableIntegrationMBeanExport`

You can now enable the `IntegrationMBeanExporter` with the `@EnableIntegrationMBeanExport` annotation in a `@Configuration` class. For more information, see MBean Exporter.

`ChannelSecurityInterceptorFactoryBean`

`ChannelSecurityInterceptorFactoryBean` now supports configuration of Spring Security for message channels that use `@Configuration` classes. For more information, see Security in Spring Integration.

Redis Command Gateway

The Redis support now provides the `<outbound-gateway>` component to perform generic Redis commands by using the `RedisConnection#execute` method. For more information, see Redis Outbound Command Gateway.

`RedisLockRegistry` and `GemfireLockRegistry`

The `RedisLockRegistry` and `GemfireLockRegistry` are now available to support global locks visible to multiple application instances and servers. These can be used with aggregating message handlers across multiple application instances such that group release occurs on only one instance. For more information, see Redis Lock Registry, Gemfire Lock Registry, and Aggregator.

`@Poller`

Annotation-based messaging configuration can now have a `poller` attribute. This means that methods annotated with `@ServiceActivator`, `@Aggregator`, and similar annotations can now use an `inputChannel` that is a reference to a `PollableChannel`. For more information, see Annotation Support.

`@InboundChannelAdapter` and `SmartLifecycle` for Annotated Endpoints

We added the `@InboundChannelAdapter` method annotation. It is an analogue of the `<int:inbound-channel-adapter>` XML component. In addition, all messaging annotations now provide `SmartLifecycle` options. For more information, see Annotation Support.

Twitter Search Outbound Gateway

We added a new twitter endpoint: `<int-twitter-search-outbound-gateway/>`. Unlike the search inbound adapter, which polls by using the same search query each time, the outbound gateway allows on-demand customized queries. For more information, see Spring Integration Social Twitter.

Gemfire Metadata Store

We added the `GemfireMetadataStore`, letting it be used, for example, in an `AbstractPersistentAcceptOnceFileListFilter` implementation in a multiple application instance or server environment. For more information, see Metadata Store, Reading Files, FTP Inbound Channel Adapter, and SFTP Inbound Channel Adapter.

`@BridgeFrom` and `@BridgeTo` Annotations

We introduced `@BridgeFrom` and `@BridgeTo` `@Bean` method annotations to mark `MessageChannel` beans in `@Configuration` classes. For more information, see Annotation Support.

Meta-messaging Annotations

Messaging annotations (`@ServiceActivator`, `@Router`, `@MessagingGateway`, and others) can now be configured as meta-annotations for user-defined messaging annotations. In addition, the user-defined annotations can have the same attributes (`inputChannel`, `@Poller`, `autoStartup`, and others). For more information, see Annotation Support.

J.11.2. General Changes

This section describes general changes from version 3.0 to version 4.0.

Requires Spring Framework 4.0

We moved the core messaging abstractions (`Message`, `MessageChannel`, and others) to the Spring Framework `spring-messaging` module. Developers who reference these classes directly in their code need to make changes, as described in the first section of the 3.0 to 4.0 Migration Guide.

Header Type for XPath Header Enricher

We introduced the `header-type` attribute for the `header` child element of the `<int-xml:xpath-header-enricher>`. This attribute provides the target type for the header value (to which the result of the XPath expression evaluation is converted). For more information see XPath Header Enricher.

Object To JSON Transformer: Node Result

We introduced the `result-type` attribute for the `<int:object-to-json-transformer>`. This attribute provides the target type for the result of mapping an object to JSON. It supports `STRING` (the default) and `NODE`. For more information see Since version 3.0, Spring Integration also provides a built-in `#xpath` SpEL function for use in expressions..

JMS Header Mapping

The `DefaultJmsHeaderMapper` now maps an incoming `JMSPriority` header to the Spring Integration `priority` header. Previously, `priority` was only considered for outbound messages. For more information, see Mapping Message Headers to and from JMS Message.

JMS Outbound Channel Adapter

The JMS outbound channel adapter now supports the `session-transacted` attribute (default: `false`). Previously, you had to inject a customized `JmsTemplate` to use transactions. See Outbound Channel Adapter.

JMS Inbound Channel Adapter

The JMS inbound channel adapter now supports the `session-transacted` attribute (default: `false`). Previously, you had to inject a customized `JmsTemplate` to use transactions. The adapter allowed

'transacted' in the `acknowledgeMode`, which was incorrect and didn't work. This value is no longer allowed. See Inbound Channel Adapter.

Datatype Channels

You can now specify a `MessageConverter` to be used when converting (if necessary) payloads to one of the accepted `datatype` instances in a Datatype channel. For more information, see Datatype Channel Configuration.

Simpler Retry Advice Configuration

We added simplified namespace support to configure a `RequestHandlerRetryAdvice`. For more information, see Configuring the Retry Advice.

Correlation Endpoint: Time-based Release Strategy

We added the mutually exclusive `group-timeout` and `group-timeout-expression` attributes to `<int:aggregator>` and `<int:resequencer>`. These attributes allow forced completion of a partial `MessageGroup`, provided the `ReleaseStrategy` does not release a group and no further messages arrive within the time specified. For more information, see Configuring an Aggregator with XML.

Redis Metadata Store

The `RedisMetadataStore` now implements `ConcurrentMetadataStore`, letting it be used, for example, in an `AbstractPersistentAcceptOnceFileListFilter` implementation in a multiple application instance or server environment. For more information, see Redis Metadata Store, Reading Files, FTP Inbound Channel Adapter, and SFTP Inbound Channel Adapter.

`JdbcChannelMessageStore` and `PriorityChannel`

T`JdbcChannelMessageStore` now implements `PriorityCapableChannelMessageStore`, letting it be used as a `message-store` reference for `priority-queue` instances. For more information, see Backing Message Channels.

AMQP Endpoints Delivery Mode

Spring AMQP, by default, creates persistent messages on the broker. You can override this behavior by setting the `amqp_deliveryMode` header or customizing the mappers. We added a convenient `default-delivery-mode` attribute to the adapters to provide easier configuration of this important setting. For more information, see Outbound Channel Adapter and Outbound Gateway.

FTP Timeouts

The `DefaultFtpSessionFactory` now exposes the `connectTimeout`, `defaultTimeout`, and `dataTimeout` properties, avoiding the need to subclass the factory to set these common properties. The `postProcess*` methods are still available for more advanced configuration. See FTP Session Factory for more information.

Twitter: `StatusUpdatingMessageHandler`

The `StatusUpdatingMessageHandler` (`<int-twitter:outbound-channel-adapter>`) now supports the `tweet-data-expression` attribute to build a `org.springframework.social.twitter.api.TweetData` object for updating the timeline status. This feature allows, for example, attaching an image. See Spring Integration Social Twitter for more information.

JPA Retrieving Gateway: `id-expression`

We introduced the `id-expression` attribute for `<int-jpa:retrieving-outbound-gateway>` to perform `EntityManager.find(Class entityClass, Object primaryKey)`. See Retrieving Outbound Gateway for more information.

TCP Deserialization Events

When one of the standard deserializers encounters a problem decoding the input stream to a message, it now emits a `TcpDeserializationExceptionEvent`, letting applications examine the data at the point at which the exception occurred. See TCP Connection Events for more information.

Messaging Annotations on `@Bean` Definitions

You can now configure messaging annotations (`@ServiceActivator`, `@Router`, `@InboundChannelAdapter`, and others) on `@Bean` definitions in `@Configuration` classes. For more information, see Annotation Support.

J.12. Changes Between 2.2 and 3.0

See the Migration Guide for important changes that might affect your applications. You can find migration guides for all versions back to 2.1 on the wiki.

J.12.1. New Components

Version 3.0 added a number of new components.

HTTP Request Mapping

The HTTP module now provides powerful request mapping support for inbound endpoints. We replaced the `UriPathHandlerMapping` class with `IntegrationRequestMappingHandlerMapping`, which is registered under the bean name of `integrationRequestMappingHandlerMapping` in the application context. Upon parsing of the HTTP inbound endpoint, either a new `IntegrationRequestMappingHandlerMapping` bean is registered or an existing bean is reused. To achieve flexible request mapping configuration, Spring Integration provides the `<request-mapping/>` child element for `<http:inbound-channel-adapter/>` and the `<http:inbound-gateway/>`. Both HTTP inbound endpoints are now fully based on the request mapping infrastructure that was introduced with Spring MVC 3.1. For example, multiple paths are supported on a single inbound endpoint. For more information see HTTP Namespace Support.

Spring Expression Language (SpEL) Configuration

We added a new `IntegrationEvaluationContextFactoryBean` to allow configuration of custom `PropertyAccessor` implementations and functions for use in SpEL expressions throughout the framework. For more information, see Spring Expression Language (SpEL).

SpEL Functions Support

To customize the SpEL `EvaluationContext` with static `Method` functions, we introduced the `<spel-function/>` component. We also added two built-in functions: `#jsonPath` and `#xpath`. For more information, see SpEL Functions.

SpEL PropertyAccessors Support

To customize the SpEL `EvaluationContext` with `PropertyAccessor` implementations, we added the `<spel-property-accessors/>` component. For more information, see Property Accessors.

Redis: New Components

We added a new Redis-based `MetadataStore` implementation. You can use the `RedisMetadataStore` to maintain the state of a `MetadataStore` across application restarts. This new `MetadataStore` implementation can be used with adapters, such as:

- Twitter inbound adapters
- Feed inbound channel adapter

We added new queue-based components. We added the `<int-redis:queue-inbound-channel-adapter/>` and `<int-redis:queue-outbound-channel-adapter/>` components to perform 'right pop' and 'left push' operations, respectively, on a Redis List.

For more information, “see Redis Support”.

Header Channel Registry

You can now instruct the framework to store reply channels and error channels in a registry for later resolution. This is useful for cases where the `replyChannel` or `errorChannel` might be lost (for example, when serializing a message). See Header Enricher for more information.

MongoDB support: New `ConfigurableMongoDbMessageStore`

In addition to the existing `eMongoDbMessageStore`, we introduced a new `ConfigurableMongoDbMessageStore`. This provides a more robust and flexible implementation of `MessageStore` for MongoDB. It does not have backward compatibility with the existing store, but we recommend using it for new applications. Existing applications can use it, but messages in the old store are not available. See MongoDb Support for more information.

Syslog Support

Building on the 2.2 `SyslogToMapTransformer`, Spring Integration 3.0 introduces `UDP` and `TCP` inbound channel adapters especially tailored for receiving SYSLOG messages. For more information, see

Syslog Support.

`tail` Support

We added file inbound channel adapters that use the `tail` command to generate messages when lines are added to the end of text files. See 'tail'ing Files.

JMX Support

We added `<int-jmx:tree-polling-channel-adapter/>`. This adapter queries the JMX MBean tree and sends a message with a payload that is the graph of objects that match the query. By default, the MBeans are mapped to primitives and simple Objects (such as `Map`, `List`, and arrays). It permits simple transformation to, for example, JSON.

The `IntegrationMBeanExporter` now allows the configuration of a custom `ObjectNamingStrategy` by using the `naming-strategy` attribute.

For more information, see JMX Support.

TCP/IP Connection Events and Connection Management

`TcpConnection` instances now emit `ApplicationEvent` instances (specifically `TcpConnectionEvent` instances) when connections are opened or closed or when an exception occurs. This change lets applications be informed of changes to TCP connections by using the normal Spring `ApplicationListener` mechanism.

We renamed `AbstractTcpConnection` to `TcpConnectionSupport`. Custom connections that are subclasses of this class can use its methods to publish events. Similarly, we renamed `AbstractTcpConnectionInterceptor` to `TcpConnectionInterceptorSupport`.

In addition, we added `<int-ip:tcp-connection-event-inbound-channel-adapter/>`. By default, this adapter sends all `TcpConnectionEvent` instances to a `Channel`.

Further, the TCP connection factories now provide a new method called `getOpenConnectionIds()`, which returns a list of identifiers for all open connections. It lets applications broadcast to all open connections, among other uses.

Finally, the connection factories also provide a new method called `closeConnection(String connectionId)`, which lets applications explicitly close a connection by using its ID.

For more information see TCP Connection Events.

Inbound Channel Adapter Script Support

The `<int:inbound-channel-adapter/>` now supports using `<expression/>` and `<script/>` child elements to create a `MessageSource`. See Channel Adapter Expressions and Scripts.

Content Enricher: Headers Enrichment Support

The content enricher now provides configuration for `<header/>` child elements, to enrich the outbound message with headers based on the reply message from the underlying message flow. For

more information see Payload Enricher.

J.12.2. General Changes

This section describes general changes from version 2.2 to version 3.0.

Message ID Generation

Previously, message IDs were generated by using the JDK `UUID.randomUUID()` method. With this release, the default mechanism has been changed to use a more efficient and significantly faster algorithm. In addition, we added the ability to change the strategy used to generate message IDs. For more information see Message ID Generation.

"<gateway>" Changes

You can now set common headers across all gateway methods, and we added more options for adding information to the message about which method was invoked.

You can now entirely customize the way that gateway method calls are mapped to messages.

The `GatewayMethodMetadata` is now a public class. It lets you programmatically configure the `GatewayProxyFactoryBean` from Java.

For more information, see Messaging Gateways.

HTTP Endpoint Changes

- **Outbound Endpoint `encode-uri`**: `<http:outbound-gateway/>` and `<http:outbound-channel-adapter/>` now provide an `encode-uri` attribute to allow disabling the encoding of the URI object before sending the request.
- **Inbound Endpoint `merge-with-default-converters`**: `<http:inbound-gateway/>` and `<http:inbound-channel-adapter/>` now have a `merge-with-default-converters` attribute to include the list of default `HttpMessageConverter` instances after the custom message converters.
- **`If-Modified-Since` and `If-Unmodified-Since` HTTP Headers**: Previously, the `If-Modified-Since` and `If-Unmodified-Since` HTTP headers were incorrectly processed within from and to HTTP headers mapped in the `DefaultHttpHeaderMapper`. Now, in addition to correcting that issue, `DefaultHttpHeaderMapper` provides date parsing from formatted strings for any HTTP headers that accept date-time values.
- **Inbound Endpoint Expression Variables**: In addition to the existing `#requestParams` and `#pathVariables`, the `<http:inbound-gateway/>` and `<http:inbound-channel-adapter/>` now support additional useful variables: `#matrixVariables`, `#requestAttributes`, `#requestHeaders`, and `#cookies`. These variables are available in both payload and header expressions.
- **Outbound Endpoint 'uri-variables-expression'**: HTTP outbound endpoints now support the `uri-variables-expression` attribute to specify an `Expression` to evaluate a `Map` for all URI variable placeholders within URL template. This allows selection of a different map of expressions based on the outgoing message.

For more information, see HTTP Support.

Jackson Support (JSON)

- A new abstraction for JSON conversion has been introduced. Implementations for Jackson 1.x and Jackson 2 are currently provided, with the version being determined by presence on the classpath. Previously, only Jackson 1.x was supported.
- The `ObjectToJsonTransformer` and `JsonToObjectTransformer` now emit/consume headers containing type information.

For more information, see "JSON Transformers" in Transformer.

Chain Elements `id` Attribute

Previously, the `id` attribute for elements within a `<chain>` was ignored and, in some cases, disallowed. Now, the `id` attribute is allowed for all elements within a `<chain>`. The bean names of chain elements is a combination of the surrounding chain's `id` and the `id` of the element itself. For example: 'myChain$child.myTransformer.handler'. For more information see, Message Handler Chain.

Aggregator 'empty-group-min-timeout' property

The `AbstractCorrelatingMessageHandler` provides a new property called `empty-group-min-timeout` to allow empty group expiry to run on a longer schedule than expiring partial groups. Empty groups are not removed from the `MessageStore` until they have not been modified for at least this number of milliseconds. For more information, see Configuring an Aggregator with XML.

Persistent File List Filters (file, (S)FTP)

New `FileListFilter` implementations that use a persistent `MetadataStore` are now available. You can use these to prevent duplicate files after a system restart. See Reading Files, FTP Inbound Channel Adapter, and SFTP Inbound Channel Adapter for more information.

Scripting Support: Variables Changes

We introduced a new `variables` attribute for scripting components. In addition, variable bindings are now allowed for inline scripts. See Groovy support and Scripting Support for more information.

Direct Channel Load Balancing configuration

Previously, when configuring `LoadBalancingStrategy` on the channel's `dispatcher` child element, the only available option was to use a pre-defined enumeration of values which did not let developers set a custom implementation of the `LoadBalancingStrategy`. You can now use `load-balancer-ref` to provide a reference to a custom implementation of the `LoadBalancingStrategy`. For more information, see `DirectChannel`.

PublishSubscribeChannel Behavior

Previously, sending to a <publish-subscribe-channel/> that had no subscribers would return a `false` result. If used in conjunction with a `MessagingTemplate`, this would result in an exception being thrown. Now, the `PublishSubscribeChannel` has a property called `minSubscribers` (default: `0`). If the message is sent to at least the minimum number of subscribers, the send operation is deemed to be

successful (even if the number is zero). If an application expects to get an exception under these conditions, set the minimum subscribers to at least 1.

FTP, SFTP and FTPS Changes

The FTP, SFTP and FTPS endpoints no longer cache sessions by default.

We removed the deprecated `cached-sessions` attribute from all endpoints. Previously, the embedded caching mechanism controlled by this attribute's value did not provide a way to limit the size of the cache, which could grow indefinitely. Release 2.1 introduced `CachingConnectionFactory`, and it became the preferred (and is now the only) way to cache sessions.

`CachingConnectionFactory` now provides a new method: `resetCache()`. This method immediately closes idle sessions and causes in-use sessions to be closed as and when they are returned to the cache.

The `DefaultSftpSessionFactory` (in conjunction with a `CachingSessionFactory`) now supports multiplexing channels over a single SSH connection (SFTP Only).

FTP, SFTP and FTPS Inbound Adapters

Previously, there was no way to override the default filter used to process files retrieved from a remote server. The `filter` attribute determines which files are retrieved, but the `FileReadingMessageSource` uses an `AcceptOnceFileListFilter`. This means that, if a new copy of a file is retrieved with the same name as a previously copied file, no message was sent from the adapter.

With this release, a new attribute `local-filter` lets you override the default filter (for example, with an `AcceptAllFileListFilter` or some other custom filter).

If you want the behavior of the `AcceptOnceFileListFilter` to be maintained across JVM executions, you can now configure a custom filter that retains state, perhaps on the file system.

Inbound channel adapters now support the `preserve-timestamp` attribute, which sets the local file modified timestamp to the timestamp from the server (default: `false`).

FTP, SFTP, and FTPS Gateways

The gateways now support the `mv` command, enabling the renaming of remote files.

The gateways now support recursive `ls` and `mget` commands, enabling the retrieval of a remote file tree.

The gateways now support `put` and `mput` commands, enabling sending files to the remote server.

The `local-filename-generator-expression` attribute is now supported, enabling the naming of local files during retrieval. By default, the same name as the remote file is used.

The `local-directory-expression` attribute is now supported, enabling the naming of local directories during retrieval (based on the remote directory).

Remote File Template

A new higher-level abstraction (`RemoteFileTemplate`) is provided over the `Session` implementations used by the FTP and SFTP modules. While it is used internally by endpoints, you can also use this abstraction programmatically. Like all Spring `*Template` implementations, it reliably closes the underlying session while allowing low level access to the session.

For more information, see FTP/FTPS Adapters and SFTP Adapters.

'requires-reply' Attribute for Outbound Gateways

All outbound gateways (such as `<jdbc:outbound-gateway/>` or `<jms:outbound-gateway/>`) are designed for 'request-reply' scenarios. A response is expected from the external service and is published to the `reply-channel` or the `replyChannel` message header. However, there are some cases where the external system might not always return a result (for example, a `<jdbc:outbound-gateway/>` when a SELECT ends with an empty `ResultSet` or perhaps a one-way web service). Consequently, developers needed an option to configure whether or not a reply is required. For this purpose, we introduced the `requires-reply` attribute for outbound gateway components. In most cases, the default value for `requires-reply` is `true`. If there is no result, a `ReplyRequiredException` is thrown. Changing the value to `false` means that, if an external service does not return anything, the message flow ends at that point, similar to an outbound channel adapter.

The WebService outbound gateway has an additional attribute called `ignore-empty-responses`. It is used to treat an empty `String` response as if no response were received. By default, it is `true`, but you can set it to `false` to allow the application to receive an empty `String` in the reply message payload. When the attribute is `true`, an empty string is treated as no response for the purposes of the `requires-reply` attribute. By default, `requires-reply` is false for the WebService outbound gateway.

Note that the `requiresReply` property was previously present but set to `false` in the `AbstractReplyProducingMessageHandler`, and there was no way to configure it on outbound gateways by using the XML namespace.

Previously, a gateway receiving no reply would silently end the flow (with a DEBUG log message). By default, with this change, an exception is now thrown by most gateways. To revert to the previous behavior, set `requires-reply` to `false`.

AMQP Outbound Gateway Header Mapping

Previously, the <int-amqp:outbound-gateway/> mapped headers before invoking the message converter, and the converter could overwrite headers such as `content-type`. The outbound adapter maps the headers after the conversion, which means headers like `content-type` from the outbound `Message` (if present) are used.

Starting with this release, the gateway now maps the headers after the message conversion, consistent with the adapter. If your application relies on the previous behavior (where the converter's headers overrode the mapped headers), you either need to filter those headers (before the message reaches the gateway) or set them appropriately. The headers affected by the `SimpleMessageConverter` are `content-type` and `content-encoding`. Custom message converters may set

other headers.

Stored Procedure Components Improvements

For more complex database-specific types not supported by the standard `CallableStatement.getObject` method, we introduced two new additional attributes to the `<sql-parameter-definition/>` element with OUT-direction:

- `type-name`
- `return-type`

The `row-mapper` attribute of the stored procedure inbound channel adapter `<returning-resultset/>` child element now supports a reference to a `RowMapper` bean definition. Previously, it contained only a class name (which is still supported).

For more information, see Stored Procedures.

Web Service Outbound URI Configuration

The web service outbound gateway 'uri' attribute now supports `<uri-variable/>` substitution for all URI schemes supported by Spring Web Services. For more information, see Outbound URI Configuration.

Redis Adapter Changes

The Redis inbound channel adapter can now use a `null` value for the `serializer` property, with the raw data being the message payload.

The Redis outbound channel adapter now has the `topic-expression` property to determine the Redis topic for the `Message` at runtime.

The Redis inbound channel adapter, in addition to the existing `topics` attribute, now has the `topic-patterns` attribute.

For more information, see Redis Support.

Advising Filters

Previously, when a `<filter/>` had a `<request-handler-advice-chain/>`, the discard action was all performed within the scope of the advice chain (including any downstream flow on the `discard-channel`). The filter element now has an attribute called `discard-within-advice` (default: `true`) to allow the discard action to be performed after the advice chain completes. See Advising Filters.

Advising Endpoints using Annotations

Request handler advice chains can now be configured using annotations. See Advising Endpoints Using Annotations.

ObjectToStringTransformer Improvements

This transformer now correctly transforms `byte[]` and `char[]` payloads to `String`. For more

information, see Transformer.

JPA Support Changes

Payloads to persist or merge can now be of type `java.lang.Iterable`.

In that case, each object returned by the `Iterable` is treated as an entity and persisted or merged by using the underlying `EntityManager`. Null values returned by the iterator are ignored.

The JPA adapters now have additional attributes to optionally flush and clear entities from the associated persistence context after performing persistence operations.

Retrieving gateways had no mechanism to specify the first record to be retrieved, which is a common use case. The retrieving gateways now support specifying this parameter by adding the `first-result` and `first-result-expression` attributes to the gateway definition. For more information, see Retrieving Outbound Gateway.

The JPA retrieving gateway and inbound adapter now have an attribute to specify the maximum number of results in a result set as an expression. In addition, we introduced the `max-results` attribute to replace `max-number-of-results`, which has been deprecated. `max-results` and `max-results-expression` are used to provide the maximum number of results or an expression to compute the maximum number of results, respectively, in the result set.

For more information, see JPA Support.

Delayer: delay expression

Previously, the `<delayer>` provided a `delay-header-name` attribute to determine the delay value at runtime. In complex cases, the `<delayer>` had to be preceded with a `<header-enricher>`. Spring Integration 3.0 introduced the `expression` attribute and `expression` child element for dynamic delay determination. The `delay-header-name` attribute is now deprecated, because you can specify the header evaluation in the `expression`. In addition, we introduced the `ignore-expression-failures` to control the behavior when an expression evaluation fails. For more information, see Delayer.

JDBC Message Store Improvements

Spring Integration 3.0 adds a new set of DDL scripts for MySQL version 5.6.4 and higher. Now MySQL supports fractional seconds and is thus improving the FIFO ordering when polling from a MySQL-based message store. For more information, see The Generic JDBC Message Store.

IMAP Idle Connection Exceptions

Previously, if an IMAP idle connection failed, it was logged, but there was no mechanism to inform an application. Such exceptions now generate `ApplicationEvent` instances. Applications can obtain these events by using an `<int-event:inbound-channel-adapter>` or any `ApplicationListener` configured to receive an `ImapIdleExceptionEvent` (or one of its super classes).

Message Headers and TCP

The TCP connection factories now enable the configuration of a flexible mechanism to transfer selected headers (as well as the payload) over TCP. A new `TcpMessageMapper` enables the selection of

the headers, and you need to configure an appropriate serializer or deserializer to write the resulting `Map` to the TCP stream. We added a `MapJsonSerializer` as a convenient mechanism to transfer headers and payload over TCP. For more information, see Transferring Headers.

JMS Message Driven Channel Adapter

Previously, when configuring a `<message-driven-channel-adapter/>`, if you wished to use a specific `TaskExecutor`, you had to declare a container bean and provide it to the adapter by setting the `container` attribute. We added the `task-executor`, letting it be set directly on the adapter. This is in addition to several other container attributes that were already available.

RMI Inbound Gateway

The RMI Inbound Gateway now supports an `error-channel` attribute. See Inbound RMI.

`XsltPayloadTransformer`

You can now specify the transformer factory class name by setting the `transformer-factory-class` attribute. See `XsltPayloadTransformer`.

J.13. Changes between 2.1 and 2.2

See the Migration Guide for important changes that might affect your applications. You can find migration guides for all versions back to 2.1 on the wiki.

J.13.1. New Components

Version 2.2 added a number of new components.

`RedisStore` Inbound and Outbound Channel Adapters

Spring Integration now has `RedisStore` Inbound and Outbound Channel Adapters, letting you write and read `Message` payloads to and from Redis collections. For more information, see RedisStore Outbound Channel Adapter and Redis Store Inbound Channel Adapter.

MongoDB Inbound and Outbound Channel Adapters

Spring Integration now has MongoDB inbound and outbound channel adapters, letting you write and read `Message` payloads to and from a MongoDB document store. For more information, see MongoDB Outbound Channel Adapter and MongoDB Inbound Channel Adapter.

JPA Endpoints

Spring Integration now includes components for the Java Persistence API (JPA) for retrieving and persisting JPA entity objects. The JPA Adapter includes the following components:

- Inbound channel adapter
- Outbound channel adapter
- Updating outbound gateway

- Retrieving outbound gateway

For more information, see JPA Support.

J.13.2. General Changes

This section describes general changes from version 2.1 to version 2.2.

Spring 3.1 Used by Default

Spring Integration now uses Spring 3.1.

Adding Behavior to Endpoints

The ability to add an `<advice-chain/>` to a poller has been available for some time. However, the behavior added by this affects the entire integration flow. It did not address the ability to add (for example) retry to an individual endpoint. The 2.2 release introduced the `<request-handler-advice-chain/>` to many endpoints.

In addition, we added three standard advice classes for this purpose:

- `MessageHandlerRetryAdvice`
- `MessageHandlerCircuitBreakerAdvice`
- `ExpressionEvaluatingMessageHandlerAdvice`

For more information, see Adding Behavior to Endpoints.

Transaction Synchronization and Pseudo Transactions

Pollers can now participate in Spring's Transaction Synchronization feature. This allows for synchronizing such operations as renaming files by an inbound channel adapter, depending on whether the transaction commits or rolls back.

In addition, you can enable these features when no "real" transaction is present, by means of a `PseudoTransactionManager`.

For more information, see Transaction Synchronization.

File Adapter: Improved File Overwrite and Append Handling

When using the file outbound channel adapter or the file outbound gateway, you can use a new `mode` property. Prior to Spring Integration 2.2, target files were replaced when they existed. Now you can specify the following options:

- `REPLACE` (default)
- `APPEND`
- `FAIL`
- `IGNORE`

For more information, see Dealing with Existing Destination Files.

Reply-Timeout Added to More Outbound Gateways

The XML Namespace support adds the reply-timeout attribute to the following outbound gateways:

- AMQP Outbound Gateway
- File Outbound Gateway
- FTP Outbound Gateway
- SFTP Outbound Gateway
- WS Outbound Gateway

Spring-AMQP 1.1

Spring Integration now uses Spring AMQP 1.1. This enables several features to be used within a Spring Integration application, including the following:

- A fixed reply queue for the outbound gateway
- HA (mirrored) queues
- Publisher confirmations
- Returned messages
- Support for dead letter exchanges and dead letter queues

JDBC Support - Stored Procedures Components

SpEL Support

When using the stored procedure components of the Spring Integration JDBC Adapter, you can now provide stored procedure names or stored function names by using the Spring Expression Language (SpEL).

Doing so lets you specify the stored procedures to be invoked at runtime. For example, you can provide stored procedure names that you would like to execute through message headers. For more information, see Stored Procedures.

JMX Support

The stored procedure components now provide basic JMX support, exposing some of their properties as MBeans:

- Stored procedure name
- Stored procedure name expression
- `JdbcCallOperations` cache statistics

JDBC Support: Outbound Gateway

When you use the JDBC outbound gateway, the update query is no longer mandatory. You can now

provide only a select query by using the request message as a source of parameters.

JDBC Support: Channel-specific Message Store Implementation

We added a new message channel-specific message store implementation, providing a more scalable solution using database-specific SQL queries. For more information, see Backing Message Channels.

Orderly Shutdown

We added a method called `stopActiveComponents()` to the `IntegrationMBeanExporter`. It allows a Spring Integration application to be shut down in an orderly manner, disallowing new inbound messages to certain adapters and waiting for some time to allow in-flight messages to complete.

JMS Outbound Gateway Improvements

You can now configure the JMS outbound gateway to use a `MessageListener` container to receive replies. Doing so can improve performance of the gateway.

`ObjectToJsonTransformer`

By default, the `ObjectToJsonTransformer` now sets the `content-type` header to `application/json`. For more information, see Transformer.

HTTP Support

Java serialization over HTTP is no longer enabled by default. Previously, when setting an `expected-response-type` on a `Serializable` object, the `Accept` header was not properly set up. We updated the `SerializingHttpMessageConverter` to set the `Accept` header to `application/x-java-serialized-object`. However, because this could cause incompatibility with existing applications, we decided to no longer automatically add this converter to the HTTP endpoints.

If you wish to use Java serialization, you need to add the `SerializingHttpMessageConverter` to the appropriate endpoints by using the `message-converters` attribute (when you use XML configuration) or by using the `setMessageConverters()` method (in Java).

Alternatively, you may wish to consider using JSON instead. It is enabled by having `Jackson` on the classpath.

J.14. Changes between 2.0 and 2.1

See the Migration Guide for important changes that might affect your applications.

J.14.1. New Components

Version 2.1 added a number of new components.

JSR-223 Scripting Support

In Spring Integration 2.0, we added support for Groovy. With Spring Integration 2.1, we expanded

support for additional languages substantially by implementing support for JSR-223 ("Scripting for the Java™ Platform"). Now you have the ability to use any scripting language that supports JSR-223 including:

- Javascript
- Ruby and JRuby
- Python and Jython
- Groovy

For further details, see Scripting Support.

GemFire Support

Spring Integration provides support for GemFire by providing inbound adapters for entry and continuous query events, an outbound adapter to write entries to the cache, and `MessageStore` and `MessageGroupStore` implementations. Spring integration leverages the Spring Gemfire project, providing a thin wrapper over its components.

For further details, see Pivotal GemFire and Apache Geode Support.

AMQP Support

Spring Integration 2.1 added several channel adapters for receiving and sending messages by using the Advanced Message Queuing Protocol (AMQP). Furthermore, Spring Integration also provides a point-to-point message channel and a publish-subscribe message channel, both of which are backed by AMQP Exchanges and Queues.

For further details, see AMQP Support.

MongoDB Support

As of version 2.1, Spring Integration provides support for MongoDB by providing a MongoDB-based `MessageStore`.

For further details, see MongoDb Support.

Redis Support

As of version 2.1, Spring Integration supports Redis, an advanced key-value store, by providing a Redis-based `MessageStore` as well as publish-subscribe messaging adapters.

For further details, see Redis Support.

Support for Spring's Resource abstraction

In version 2.1, we introduced a new resource inbound channel adapter that builds upon Spring's resource abstraction to support greater flexibility across a variety of actual types of underlying resources, such as a file, a URL, or a classpath resource. Therefore, it is similar to but more generic than the file inbound channel adapter.

For further details, see Resource Inbound Channel Adapter.

Stored Procedure Components

With Spring Integration 2.1, the `JDBC` Module also provides stored procedure support by adding several new components, including inbound and outbound channel adapters and an outbound gateway. The stored procedure support leverages Spring's `SimpleJdbcCall` class and consequently supports stored procedures for:

- Apache Derby
- DB2
- MySQL
- Microsoft SQL Server
- Oracle
- PostgreSQL
- Sybase

The stored procedure components also support SQL functions for the following databases:

- MySQL
- Microsoft SQL Server
- Oracle
- PostgreSQL

For further details, see Stored Procedures.

XPath and XML Validating Filter

Spring Integration 2.1 provides a new XPath-based message filter. It is part of the `XML` module. The XPath filter lets you filter messages by using XPath Expressions. We also added documentation for the XML validating filter.

For more details, see Using the XPath Filter and XML Validating Filter.

Payload Enricher

Since Spring Integration 2.1, we added the payload enricher. A payload enricher defines an endpoint that typically passes a `Message` to the exposed request channel and then expects a reply message. The reply message then becomes the root object for evaluation of expressions to enrich the target payload.

For further details, see Payload Enricher.

FTP and SFTP Outbound Gateways

Spring Integration 2.1 provides two new outbound gateways to interact with remote File Transfer Protocol (FTP) or Secure File Transfer Protocol (SFT) servers. These two gateways let you directly

execute a limited set of remote commands.

For instance, you can use these outbound gateways to list, retrieve, and delete remote files and have the Spring Integration message flow continue with the remote server's response.

For further details, see FTP Outbound Gateway and SFTP Outbound Gateway.

FTP Session Caching

As of version 2.1, we have exposed more flexibility with regards to session management for remote file adapters (for example, FTP, SFTP, and others).

Specifically, we deprecated the `cache-sessions` attribute (which is available via the XML namespace support). As an alternative, we added the `sessionCacheSize` and `sessionWaitTimeout` attributes on the `CachingSessionFactory`.

For further details, see FTP Session Caching and SFTP Session Caching.

J.14.2. Framework Refactoring

We refactored the Spring Integration framework in a number of ways, all described in this section.

Standardizing Router Configuration

We standardized router parameters across all router implementations with Spring Integration 2.1 to provide a more consistent user experience.

In Spring Integration 2.1, we removed the `ignore-channel-name-resolution-failures` attribute in favor of consolidating its behavior with the `resolution-required` attribute. Also, the `resolution-required` attribute now defaults to `true`.

Starting with Spring Integration 2.1, routers no longer silently drop any messages if no default output channel was defined. This means that, by default, routers now require at least one resolved channel (if no `default-output-channel` was set) and, by default, throw a `MessageDeliveryException` if no channel was determined (or an attempt to send was not successful).

If, however, you do want to drop messages silently, you can set `default-output-channel="nullChannel"`.

With the standardization of router parameters and the consolidation of the parameters described earlier, older Spring Integration based applications may break.

For further details, see `Routers`.

XML Schemas updated to 2.1

Spring Integration 2.1 ships with an updated XML Schema (version 2.1). It provides many improvements, such as the Router standardizations discussed earlier.

From now on, developers must always declare the latest XML schema (currently version 2.1).

Alternatively, they can use the version-less schema. Generally, the best option is to use version-less namespaces, as these automatically use the latest available version of Spring Integration.

The following example declares a version-less Spring Integration namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:int="http://www.springframework.org/schema/integration"
   xsi:schemaLocation="http://www.springframework.org/schema/integration
           https://www.springframework.org/schema/integration/spring-
integration.xsd
           http://www.springframework.org/schema/beans
           https://www.springframework.org/schema/beans/spring-beans.xsd">
...
</beans>
```

The following example declares a Spring Integration namespace with an explicit version:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:int="http://www.springframework.org/schema/integration"
   xsi:schemaLocation="http://www.springframework.org/schema/integration
           https://www.springframework.org/schema/integration/spring-integration-
2.2.xsd
           http://www.springframework.org/schema/beans
           https://www.springframework.org/schema/beans/spring-beans.xsd">
...
</beans>
```

The old 1.0 and 2.0 schemas are still there. However, if an application context still references one of those deprecated schemas, the validator fails on initialization.

J.14.3. Source Control Management and Build Infrastructure

Version 2.1 introduced a number of changes to source control management and build infrastructure. This section covers those changes.

Source Code Now Hosted on Github

Since version 2.0, the Spring Integration project uses Git for version control. To increase community visibility even further, the project was moved from SpringSource hosted Git repositories to Github. The Spring Integration Git repository is located at: spring-integration.

For the project, we also improved the process of providing code contributions. Further, we ensure that every commit is peer-reviewed. In fact, core committers now follow the same process as contributors. For more details, see Contributing.

Improved Source Code Visibility with Sonar

In an effort to provide better source code visibility and consequently to monitor the quality of Spring Integration's source code, we set up an instance of Sonar. We gather metrics nightly and make them available at sonar.spring.io.

J.14.4. New Samples

For the 2.1 release of Spring Integration, we also expanded the Spring Integration Samples project and added many new samples, such as samples that cover AMQP support, a sample that showcases the new payload enricher, a sample illustrating techniques for testing Spring Integration flow fragments, and a sample for executing stored procedures against Oracle databases. For details, visit spring-integration-samples.

J.15. Changes between Versions 1.0 and 2.0

See the Migration Guide for important changes that might affect your applications.

J.15.1. Spring 3 support

Spring Integration 2.0 is built on top of Spring 3.0.5 and makes many of its features available to our users.

[[2.0-spel-support]] ===== Support for the Spring Expression Language (SpEL)

You can now use SpEL expressions within the transformer, router, filter, splitter, aggregator, service-activator, header-enricher, and many more elements of the Spring Integration core namespace as well as within various adapters. This guide includes many samples.

Conversion Service and Converter

You can now benefit from the conversion service support provided with Spring while configuring many Spring Integration components, such as a Datatype channel. See Message Channel Implementations and Service Activator. Also, the SpEL support mentioned in the previous point also relies upon the conversion service. Therefore, you can register converters once and take advantage of them anywhere you use SpEL expressions.

`TaskScheduler` and `Trigger`

Spring 3.0 defines two new strategies related to scheduling: `TaskScheduler` and `Trigger`. Spring Integration (which uses a lot of scheduling) now builds upon these. In fact, Spring Integration 1.0 had originally defined some of the components (such as `CronTrigger`) that have now been migrated into Spring 3.0's core API. Now you can benefit from reusing the same components within the entire application context (not just Spring Integration configuration). We also greatly simplified configuration of Spring Integration pollers by providing attributes for directly configuring rates,

delays, cron expressions, and trigger references. See Channel Adapter for sample configurations.

`RestTemplate` and `HttpMessageConverter`

Our outbound HTTP adapters now delegate to Spring's `RestTemplate` for executing the HTTP request and handling its response. This also means that you can reuse any custom `HttpMessageConverter` implementations. See HTTP Outbound Components for more details.

J.15.2. Enterprise Integration Pattern Additions

Also in 2.0, we have added support for even more of the patterns described in Hohpe and Woolf's Enterprise Integration Patterns book.

Message History

We now provide support for the message history pattern, letting you keep track of all traversed components, including the name of each channel and endpoint as well as the timestamp of that traversal. See Message History for more details.

Message Store

We now provide support for the message store pattern. The message store provides a strategy for persisting messages on behalf of any process whose scope extends beyond a single transaction, such as the aggregator and the resequencer. Many sections of this guide include samples of how to use a message store, as it affects several areas of Spring Integration. See Message Store, Claim Check, Message Channels, Aggregator, JDBC Support`", and Resequencer for more details.

Claim Check

We have added an implementation of the claim check pattern. The idea behind the claim check pattern is that you can exchange a message payload for a "claim ticket". This lets you reduce bandwidth and avoid potential security issues when sending messages across channels. See Claim Check for more details.

Control Bus

We have provided implementations of the control bus pattern, which lets you use messaging to manage and monitor endpoints and channels. The implementations include both a SpEL-based approach and one that runs Groovy scripts. See Control Bus and Control Bus for more details.

J.15.3. New Channel Adapters and Gateways

We have added several new channel adapters and messaging gateways in Spring Integration 2.0.

TCP and UDP Adapters

We have added channel adapters for receiving and sending messages over the TCP and UDP internet protocols. See TCP and UDP Support for more details. See also the following blog: "Using UDP and TCP Adapters in Spring Integration 2.0 M3".

Twitter Adapters

Twitter adapters provides support for sending and receiving Twitter status updates as well as direct messages. You can also perform Twitter Searches with an inbound channel adapter. See Spring Integration Social Twitter for more details.

XMPP Adapters

The new XMPP adapters support both chat messages and presence events. See XMPP Support for more details.

FTP and FTPS Adapters

Inbound and outbound file transfer support over FTP and FTPS is now available. See FTP/FTPS Adapters for more details.

SFTP Adapters

Inbound and outbound file transfer support over SFTP is now available. See SFTP Adapters for more details.

Feed Adapters

We have also added channel adapters for receiving news feeds (ATOM and RSS). See Feed Adapter for more details.

J.15.4. Other Additions

Spring Integration adds a number of other features. This section describes them.

Groovy Support

Spring Integration 2.0 added Groovy support, letting you use the Groovy scripting language to provide integration and business logic. See Groovy support for more details.

Map Transformers

These symmetrical transformers convert payload objects to and from `Map` objects. See Transformer for more details.

JSON Transformers

These symmetrical transformers convert payload objects to and from JSON. See Transformer for more details.

Serialization Transformers

These symmetrical transformers convert payload objects to and from byte arrays. They also support the serializer and deserializer strategy interfaces that Spring 3.0.5 added. See Transformer for more details.

J.15.5. Framework Refactoring

The core API went through some significant refactoring to make it simpler and more usable. Although we anticipate that the impact to developers should be minimal, you should read through this document to find what was changed. Specifically, you should read Dynamic Routers, Messaging Gateways, HTTP Outbound Components, Message, and Aggregator. If you directly depend on some of the core components (`Message`, `MessageHeaders`, `MessageChannel`, `MessageBuilder`, and others), you need to update any import statements. We restructured some packaging to provide the flexibility we needed for extending the domain model while avoiding any cyclical dependencies (it is a policy of the framework to avoid such "tangles").

J.15.6. New Source Control Management and Build Infrastructure

With Spring Integration 2.0, we switched our build environment to use Git for source control. To access our repository, visit git.springsource.org/spring-integration. We have also switched our build system to Gradle.

J.15.7. New Spring Integration Samples

With Spring Integration 2.0, we have decoupled the samples from our main release distribution. Please read the following blog to get more information: New Spring Integration Samples. We have also created many new samples, including samples for every new adapter.

J.15.8. Spring Tool Suite Visual Editor for Spring Integration

There is an amazing new visual editor for Spring Integration included within the latest version of SpringSource Tool Suite. If you are not already using STS, you can download it at Spring Tool Suite.